Formale Verifikation von Software für speicherprogrammierbare Steuerungen mittels Model Checking

Von der Carl-Friedrich-Gauß-Fakultät
Technische Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades
Doktor-Ingenieurin (Dr.-Ing.)

genehmigte Dissertation

von

Olivera Pavlović

geboren am 17. September 1977 in Leskovac, Serbien

Eingereicht am: 12. November 2009
Mündliche Prüfung am: 16. Dezember 2009

Referent: Prof. em. Dr. Hans-Dieter Ehrich
Koreferent: Prof. Dr. rer. nat. Jens Braband

2010

Danksagung

Die vorliegende Arbeit entstand im Laufe meiner Tätigkeit als Stipendiatin im Graduiertenkolleg „Rail Automation Graduate School (RA:GS!)“ bei der Siemens AG. Ich möchte mich bei all denen, die mich bei der Entstehung dieser Arbeit begleitet haben, von ganzem Herzen bedanken.

Herrn Prof. em. Dr. Hans-Dieter Ehrich, danke ich recht herzlich für die langjährige Betreuung während meiner Studien- und Promotionszeit. Herrn Prof. Dr. rer. nat. Jens Braband gilt mein Dank für die Möglichkeit des eigenverantwortlichen Bearbeitens eines äußerst interessanten Themas im industriellen Umfeld.

Bei allen Siemens-Kollegen bedanke ich mich für die nette Arbeitsatmosphäre und die stetige Unterstützung. Mein besonderer Dank geht an Ralf Pinger für die langjährige Betreuung meiner Arbeit, an Uwe Eckelmann-Wendt für die zahlreiche Fachgespräche, sowie an Stefan Gerken, Bernd Lanzendörfer und Maico Ostwald für das Korrekturlesen meiner Arbeit.

Ich bin froh, dass ich während meiner Promotionszeit ein Teil des Instituts für Informationssysteme sein durfte. Für diese nette Zeit möchte ich mich bei allen Institutskollegen ganz herzlich bedanken.

Meinen Freunden, Maja Miličić-Brandt und Sebastian Brandt, danke ich von ganzem Herzen für die Unterstützung und die zahlreichen Korrekturvorschläge.

Ohne meine Eltern und deren tiefen Glaube an ihre Kinder, wäre ich nie so weit gekommen. Bei ihnen, meinem Bruder und meiner Oma bedanke ich mich, dass ich mich zu der Person entwickeln konnte, die ich jetzt bin.

Der größte Dank gilt meinem Mann für seine ununterbrochene Unterstützung in den letzten Jahren. Ohne seine Hilfe und Ermutigung hätte ich mich nicht getraut, diesen Weg zu gehen. Bei meinem Sohn bedanke ich mich, dass er zur Welt gekommen ist und einen strahlenden Sonnenschein in mein Leben gebracht hat. Nenad und Matija, vielen Dank für Eure Geduld. Euch widme ich diese Arbeit.

Braunschweig, im Juni 2010

Olivera Pavlović

Abstract

Programmable logic controllers (PLCs) are electronic systems used for different industrial control tasks. The development of PLCs in the last years has made it possible to apply PLCs for tasks of higher complexity. Many systems based on these controllers are safety-related systems. The certification of which entails a great effort. Therefore, there is a big demand for tools to analyze PLC applications and providing a proof of correctness for these systems.

Within the scope of this thesis it is examined, how a proof of correctness of safety-related systems based on PLCs can be provided. The IEC has proposed five languages for PLC programming. The verification of one of the languages, the graphical PLC programming language Function Block Diagram (FBD), has been examined in this thesis. The specific PLC used as case study for this thesis is a Siemens SIMATIC S7.

The automation of the verification process is an important precondition for efficient involvement of the verification in the development of PLCs. For this purpose a suitable verification method is model checking. In the process each state of the model is explored and checked, whether the designated property is fulfilled in the state or not. A method for automated transformation of an FBD program into a model suitable for model checking has been proposed in this work. The application of the method to the field of rail automation is described by a case study. In the case study the specification has also been automatically transformed into an appropriate logic. Therewith the whole verification process has been fully automated.

This work has been developed within the scope of "Rail Automation Graduate School (RA:GS!)" by Siemens AG, Industry Sector at Brunswick. The gained results provide an evidence, that formal methods have the potential for commercial use in the industrial field.

Zusammenfassung

Speicherprogrammierbare Steuerungen (SPS) sind elektronische, für verschiedene Steuerungsaufgaben in industrieller Umgebung eingesetzte Systeme. Parallel zur allgemeinen Entwicklung der Technik werden auch SPS weiterentwickelt und für immer komplexere Aufgaben eingesetzt. Da heutzutage viele Systeme mit Sicherheitsverantwortung auf SPS basieren, ist es vor dem Einsatz wichtig feststellen zu können, ob sie fehlerfrei funktionieren oder nicht.

Im Rahmen dieses Dissertationsprojekts wird untersucht, wie die Korrektheit einer Anwendungsapplikation für sicherheitsrelevante Systeme auf Basis einer SPS bewiesen werden kann. Das internationale Normungsgremium für Normen im Bereich Elektrotechnik und Elektronik hat fünf Sprachen für die SPS-Programmierung vorgeschlagen. In der vorliegenden Arbeit wird das Verifikationspotenzial einer dieser Sprachen, der graphischen SPS-Programmiersprache Funktionsbaustein-Sprache (FBS), untersucht. Konkret wird hier mit der SIMATIC S7-Steuerung gearbeitet, deren entsprechende Sprache als Funktionsplan (FUP) bezeichnet ist.

Eine wichtige Voraussetzung für die effiziente Einbindung der Verifikation in die SPS-Entwicklung ist die Automatisierung des Verifikationsprozesses. Eine dafür geeignete Verifikationsmethode ist Model Checking. Dabei werden alle Zustände des Modells untersucht und es wird überprüft, ob in jedem Zustand die gewünschte Eigenschaft erfüllt ist. In dieser Arbeit wird der in einer SPS-Programmiersprache erstellte Code automatisch zum Modell transformiert. Es wird eine Fallstudie vorgestellt, mit der die Anwendung der Methode im Bereich der Eisenbahnautomatisierung beschrieben wird. In der Fallstudie wird zusätzlich auch die Spezifikation automatisch in eine geeignete Logik überführt. Damit wird der ganze Prozess der formalen Verifikation vollautomatisiert.

Diese Arbeit entstand im Rahmen des Graduiertenkollegs „Rail Automation Graduate School (RA:GS!)“ der Siemens AG, Industry Sector in Braunschweig. Die erzielten Ergebnisse weisen nach, dass formale Verifikation das Potenzial zum kommerziellen Einsatz in industrieller Umgebung besitzt.

Inhaltsverzeichnis

1 Einführung

Wir leben in einer Welt der automatischen Abläufe. Viele unserer Aktivitäten des Alltags beruhen in hohem Maße auf dem korrekten Funktionieren von Rechnersystemen. Auch wenn wir nicht persönlich oder beruflich direkt auf die Hilfe von Rechnern angewiesen sind, gibt es dutzende offensichtliche Beispiele für rechnergesteuerte Systeme in unserer Umgebung.

Eine speicherprogrammierbare Steuerungen (SPS) ist eine Hardwareplattform, die zur Steuerung einer Anlage eingesetzt werden kann. Zusammen mit der Software, die für die Projektierung von Automatisierungslösungen zuständig ist, weisen SPS ein breites Spektrum von Anwendungen auf. Sie können beispielsweise bei elektronischen Heizsystemen zu Hause, bei Seilbahnen in Skigebieten, bei elektronischen Stellwerken im Eisenbahnverkehr oder bei Schutzsystemen in Kernkraftwerken eingesetzt werden.

In der IEC 61131-3 ([Int03]) sind fünf Sprachen für die Programmierung von SPS vorgeschlagen. Zwei von diesen sind Textsprachen: AWL (Anweisungsliste) und ST (Strukturierter Text). Die restlichen drei sind grafische Sprachen: FBS (Funktionsbaustein-Sprache), KOP (Kontaktplan) und AS (Ablaufsprache). AWL ist eine Assemblersprache für eine Einoperanden Stackmaschine, die einen geringen Speicherplatzbedarf aufweist, andererseits ist die Lesbarkeit des AWL-Codes schlecht. Die zweite Textprogrammiersprache ST ist eine Pascal-ähnliche Sprache. FBS zeichnet sich dadurch aus, dass kombinatorische Logik in Blöcke und Kontrolfluß durch Sprünge zu anderen Blöcke dargestellt sind. KOP lehnt sich in der Darstellung an Stromlaufpläne an, während AS eine Art Zustandsdiagramm darstellt.

In der IEC 61131-3 sind die Programmiersprachen vorgeschlagen, ohne ihre Semantik zu beschreiben. Eine vollständige Beschreibung der Sprachen ist an die konkrete Implementierung der Sprachen für eine konkrete SPS gebunden. SIMATIC ist

eine Familie von Steuerungen, die von der Firma Siemens entwickelt wird. SIMATIC wird mit der Programmiersoftware STEP 7 programmiert. Die Programmierung folgt dann in allen fünf Sprachen aus der Norm. Spezifisch für SIMATIC ist, dass die Implementierung von FBS Funktionsplan (FUP) genannt wird und dass die Implementierung von AS S7-GRAPH genannt wird.

Heutzutage basieren viele Systeme mit Sicherheitsverantwortung auf SPS. Deswegen besteht ein hohes Interesse festzustellen, ob SPS fehlerfrei funktionieren, bevor sie für ihre Aufgabe angewendet werden. Fehlerfreiheit der SPS unterscheidet dann die Fehlerfreiheit der Hardware und die Fehlerfreiheit der Software. Hardware und Software können systematische Fehler haben. Hardware kann zusätzlich zufällige Fehler haben. Sie kann während des Betriebs durch Alterung ausfallen. Software altert dann, wenn die Hardware erneuert wird und Software nicht mehr genau auf die neue Hardware Generation paßt. Die vorliegende Arbeit befasst sich mit systematischen Fehlern in der Software.

Bei Software mit überschaubaren Grenzen lässt sich durch manuell erstellte Testfälle feststellen, ob die Software die gewünschten Eigenschaften erfüllt und frei von unerwünschten Eigenschaften ist oder nicht. Gemeinsam mit der Komplexität der Aufgaben, für die SPS eingesetzt werden, steigt aber auch die Komplexität der SPS-Software. Dadurch wird es immer schwieriger, alle kritischen Situationen durch Testfälle abzudecken und die Korrektheit der Software zu zeigen. Eine mögliche Antwort auf diese Schwierigkeiten besteht darin, Software formal zu verifizieren statt sie zu testen. Durch formale Verifikation wird nicht nur die Richtigkeit einzelner Testfälle überprüft, sondern es wird bewiesen, dass ein Modell vorgegebene Eigenschaften erfüllt.

Die effiziente Einbettung formaler Verifikation in die SPS-Entwicklung setzt voraus, dass das Verifikationsverfahren automatisch abläuft. Als eine geeignete Methode für die SPS-Verifikation ergibt sich damit das Model Checking, da diese Verifikationsmethode viele Möglichkeiten der Automatisierung bietet. Dabei werden alle Zustände des Modells untersucht und es wird überprüft, ob in jedem Zustand die gewünschten Eigenschaften gelten.

1.1 Motivation

Auf dem Gebiet SPS-Verifikation wurden in letzter Zeit erhebliche Fortschritte gemacht. Dennoch hat sich in der Industrie noch kein Verifikationsverfahren als Standard durchgesetzt. Daraus kann man schließen, dass dieses Gebiet noch viel Freiraum für neue Ergebnisse bietet. Im Gegensatz zu den anderen SPS-Sprachen wurde FBS hinsichtlich formaler Verifikation nicht ausgiebig untersucht. Erst in den letzten fünf Jahren wurden die Aktivitäten auf diesem Gebiet intensiver.

Die vorliegende Arbeit entstand im Rahmen des Graduiertenkollegs „Rail Automation Graduate School (RA:GS!)“ der Siemens AG, Industry Sector in Braunschweig. An diesem Standort wird Eisenbahnsignaltechnik entwickelt, wobei erhebliche Anforderungen an die Sicherheit gestellt werden. Seit einiger Zeit werden SPS als Basis für Systeme mit mittleren und mittlerweile auch höchsten Sicherheitsanforderungen verwendet. Nachteilig ist jedoch, dass für diese SPS keine geeigneten Werkzeuge erhältlich sind, die formal die Korrektheit und damit die fehlerfreie Ausführung der Sicherheitsfunktionen der Anwenderapplikation sicherstellen.

Systeme und Komponenten in der Eisenbahntechnik werden üblicherweise durch Simulation von manuell erstellten Testfällen getestet. Diese Methode ist erfolgreich, weil die Entwickler der Tests fachliche Kompetenz besitzen. Andererseits sind Simulationen und Tests kosten- und zeitintensiv und besonders bei großen Systemen ist es schwierig, einen kompletten Überdeckungstest zu erstellen. Seit den 80ern gibt es eine Reihe von Anwendungen formaler Verifikation in der Eisenbahntechnik. Beispielsweise wurde in [HPP+99] ein Verfahren für die formale Verifikation von der Steuerung eines Bahnübergangs vorgestellt. Viele andere Beispiele aus dem letzten Jahrzehnt findet man in der Tagungsreihe FORMS/FORMAT (Formale Techniken für Automatisierungs- und Sicherheitssysteme im Eisenbahn- und Automotivebereich), die sich dem Thema intensiv widmet. Ein neueres Beispiel findet man in [Kin08]. Dabei schlägt der Autor ein Verfahren für die Verifikation von bestimmten Achszählsystemen vor. Die vorliegende Arbeit stellt die Fortsetzung der Anwendung von formaler Verifikation in der Eisenbahntechnik dar.

Die auf diesem Forschungsgebiet erzielten Ergebnisse weisen großes Potenzial auf, in der Industrie eingesetzt zu werden. Die effiziente Entwicklung formal verifizierbarer Anwendungsapplikationen stellt einen Beitrag zu korrekter und kostengünstiger Sicherungstechnik dar und eignet sich vor allem für deren mittleres Marktsegment.

1.2 Zielsetzung

Das Ziel dieser Dissertation hat eine theoretische und eine praktische Dimension. Auf der theoretischen Seite sollte ein Verfahren für die Verifikation von graphischen FBS-Programmen entwickelt werden. Dies ist die einzige SPS-Sprache, deren Verifikation noch nicht ausreichend untersucht wurde. Theoretisch betrachtet, ist es möglich, ein FBS-Programm ins maschinenorientierte AWL-Programm zu überführen und die Verifikation darauf durchzuführen. Die aus AWL-Programmen generierten Modelle haben einen deutlich größeren Zustandsraum als Modelle, die aus FBS direkt generiert werden. Das liegt daran, da die AWL-Modelle doe komplette SPS CPU über die maschinennahe Sprache AWL zusätzlich mitbringen. Deswegen ist bei der praktischen Dimension dieser Arbeit sehr wichtig, allein mit den Formulierungen aus FBS zu arbeiten.

Auf der praktischen Seite sollte das entwickelte Verfahren auf SPS-Applikationen aus realer Industrieumgebung anwendbar sein. Dementsprechend sollte diese Methode ermöglichen, komplexe, praxisrelevante Fallbeispiele zu behandeln. Demzufolge sollte das Verifikationsverfahren eine geeignete Lösung für das Problem der Zustandsexplosion beim Model Checking bieten. Dieses Problem stellt eine große Herausforderung für Model Checking Verfahren dar und ist ein Grund dafür, dass Model Checking häufig nicht bei großen praxisrelevanten Problemstellungen eingesetzt werden kann.

Konkret wird in der vorliegenden Arbeit eine SIMATIC Steuerung betrachtet. Es wird die entsprechende FBS-Sprache in der SIMATIC-Programmiersoftware als Funktionsplan (FUP) bezeichnet. Dementsprechend wird im Folgenden von FUP-Verifikation gesprochen.

1.3 Ergebnisse

Die Hauptbeiträge der vorliegenden Arbeit sind die folgenden:

Transformation von aus FUP erzeugtes *AWL* zur textuellen Representation von *FUP*

Sowohl für die Programmierung als auch für die Analyse von SPS-Software ist hochqualifiziertes Personal notwendig. Für die Programmierung von SPS werden Exper-

ten gebraucht, da dabei Kenntnisse von SPS-spezifischen Sprachen erforderlich ist. Was die Analyse der SPS FUP-Software betrifft, gibt es immer noch keine für genaue Untersuchung geeigneten Werkzeuge. Aus diesem Grund ist es notwendig, SPS FUP-Software in eine andere, weit verbreitete, Programmiersprache (wie z.B. C) transformieren zu können, für die ausreichende Analysewerkzeuge vorhanden sind. In dieser Arbeit wird erstmalig gezeigt, dass so eine Transformation möglich ist. Es wird ein Parser konstruiert, der FUP-Programme in die C-ähnliche Sprache textFUP transformiert.

Verifikation von FUP-Programmen

FUP ist die einzige SPS-Sprache, die im Sinne der Verifikation von SPS-Software noch nicht ausreichend untersucht wurde. In dieser Arbeit wird ein Verfahren für die Verifikation von FUP-Programmen vorgestellt.

Verifikation in der Eisenbahntechnik

Als Fallstudie für diese Arbeit wurde eine auf SPS-basierende Stellwerkssoftware aus der Eisenbahntechnik benutzt. Mit diesem Dissertationsprojekt wird gezeigt, dass sich formale Verifikation auf das Gebiet der Eisenbahntechnik anwenden lässt. Darüber hinaus wird ein Tool gebaut, mit dem man FUP-Programme im Allgemeinen, also über die konkrete Fallstudie hinaus, verifizieren kann.

Komplette Automatisierung der Verifikation

Zum Zweck der Verifikation wurde in dieser Arbeit der Model Checker NuSMV benutzt. Es wurde erreicht, sowohl das SPS-Programm automatisch in NuSMV-Modelle zu transformieren, als auch die Spezifikation automatisch in temporaler Logik darzustellen. Damit wird die komplette Automatisierung des Verifikationsprozesses erreicht.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit ist folgendermaßen aufgebaut:

- Kapitel 2 bietet einen Überblick über SPS, deren Aufbau und Programmierung.

- In Kapitel 3 werden die Grundprinzipien formaler Verifikation erläutert. Anschließend wird der Stand der Technik im Bereich formaler Verifikation von SPS-Software analysiert.
- Kapitel 4 liefert einen Beitrag zur Formalisiereung von SPS. In dem Kapitel wird eine formale Beschreibung der Syntax und Semantik von FUP vorgestellt.
- Kapitel 5 enthält die Hauptbeiträge dieser Arbeit. Es wird zuerst beschrieben, wie sich AWL-Programme verifizieren lassen. Detailliert wird das Verfahren für die Verifikation von FUP-Programmen dargestellt.
- In Kapitel 6 werden das Verifikationstool sowie die Fallstudie präsentiert. Hierzu wird ein Baustein aus der Stellwerksoftware betrachtet.
- Kapitel 7 fasst die erzielten Ergebnisse zusammen und gibt einen Ausblick auf zukünftige Arbeit auf dem Forschungsgebiet.

2 Speicherprogrammierbare Steuerungen

Eine Steuerung wird benutzt, um den Arbeitsablauf eines Gerätes oder eines Prozesses zu beeinflussen. Abhängig von Eingangsgrößen für den Steuerungsalgorithmus und vom Zustand, in dem sich der zu steuernde Prozess befindet, werden Ausgangsgrößen eingestellt. Üblicherweise werden Informationen aus der Prozessumgebung durch Sensoren erfasst. Die Signale aus diesen Sensoren werden in Eingangsgrößen für die Steuerung umgewandelt. Die von der Steuerung erstellten Ausgangsgrößen werden dann mittels Aktoren an den Prozess übermittelt. Da viele Steuerungen eine Reaktion innerhalb definierter Zeit brauchen, müssen sie also echtzeitfähig sind.

Es gibt Steuerungen, deren Logik durch eine feste Verbindung der einzelnen Bauelementen festgelegt ist. Von Vorteil bei diesen festverdrahteten Steuerungen ist ihre Zuverlässigkeit, die als höher gilt als bei elektronischen Steuerungen. Nachteilig ist jedoch, dass die Steuerungen nur geringe Flexibilität besitzen. Andererseits weisen speicherprogrammierbare Steuerungen (SPS) (siehe Abbildung 2.1) die hohe Flexibilität bezüglich der Änderbarkeit ihrer Programme (Logik) auf. Nachteilig bei SPS ist, dass es nicht so einfach ist, die Korrektheit ihrer Logik auch zu beweisen.

Die fehlerfreie Funktionsweise einer SPS kann beispielsweise durch Verifikation gezeigt werden. Dabei muss sowohl die SPS-Software als auch die SPS-Hardware verifiziert werden. Der Schwerpunkt dieser Arbeit liegt in der Softwareverifikation. Bevor in den folgenden Kapiteln ein Verfahren für die formale Verifikation von SPS-Software vorgestellt wird, werden in diesem Kapitel die SPS-Grundlagen erläutert. Dementsprechend werden im Folgenden Aufbau und Arbeitsweise einer SPS beschrieben. Anschließend wird erklärt, wie eine SPS programmiert wird. Am Ende des Kapitels wird als Beispiel die Steuerung SIMATIC S7 vorgestellt.

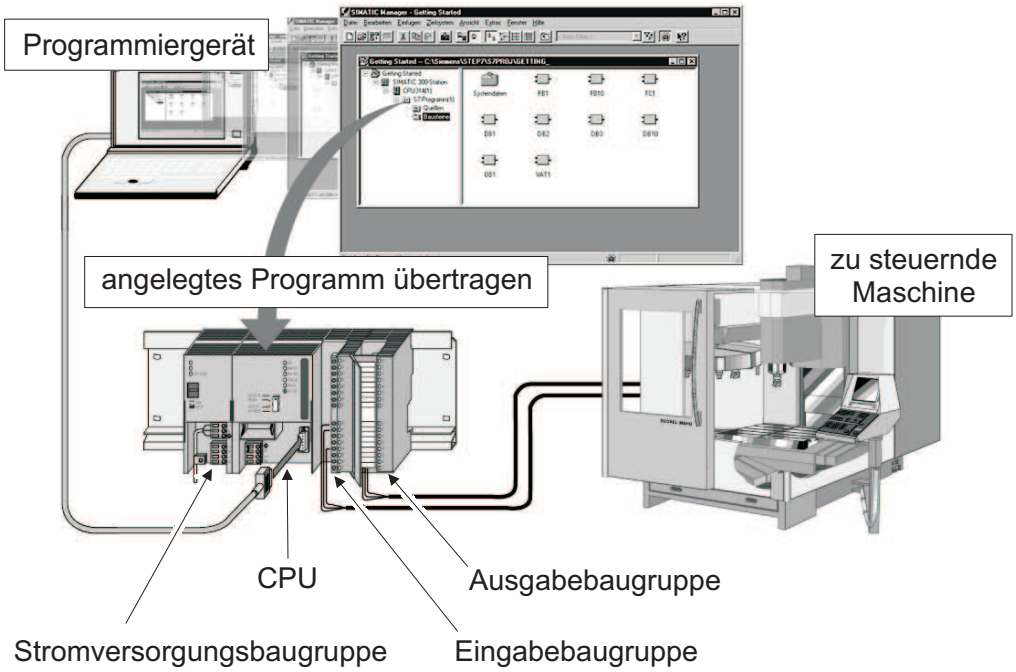


Abbildung 2.1: Eine Speicherprogrammierbare Steuerung in ihrem Umfeld ([Sie04a])

2.1 Aufbau

Abhängig vom Hersteller und vom Typ kann eine SPS kompakt oder modular aufgebaut sein. Als Beispiel wird in Abbildung 2.1 die Steuerung SIMATIC S7-300 dargestellt (siehe [Sie04a]), die modular aufgebaut ist. Die SPS besteht aus einer Stromversorgung, einer Zentraleinheit, und Ein- beziehungsweise Ausgabebaugruppen (E/A-Baugruppen). Diese Einheiten sind typisch für eine SPS mit modularer Struktur.

Der Kern einer SPS ist ihre Zentraleinheit (CPU). Die CPU ist zuständig für die Informationsverarbeitung und übernimmt die Abarbeitung des Steuerungsprogramms (siehe [Gie05]). Wie bei der zugrundeliegenden von-Neumann-Architektur benötigt sie zu diesem Zweck:

- Steuerwerk - Wie bei jedem Mikrocontroller beziehungsweise Rechner ermöglicht das Steuerwerk zusammen mit dem Betriebssystem einen Ablauf der Aktionen vom Einschalten bis zur Kommunikation mit der Umgebung.

- Arbeitsregister - Dazu gehören spezielle Register, wie: Akkumulatoren (Akkus), Klammerstack und Statuswort. Diese Registern werden in Abschnitt 2.4 am Beispiel einer konkreten Steuerung näher erläutert.

- Speicher - Dabei muss man einen Unterschied zwischen dem Ladespeicher und dem Arbeitsspeicher machen. Im Ladespeicher ist das Anwenderprogramm enthalten. Allerdings ist der Ladespeicher nicht an der Programmabarbeitung beteiligt. Dazu ist der Arbeitsspeicher zuständig. In einem seiner Bereiche enthält der Arbeitsspeicher eine Kopie der Programmbausteine aus dem Ladespeicher. Im anderen Bereich sind bestimmte SPS-typische operative Daten (Prozessabbilder, Merker, Timer, Zähler und Lokaldaten) zusammengefasst.

Wie es in Abbildung 2.1 dargestellt ist, wird eine SPS meistens mit einem Programmiergerät oder einem Rechner (PC) mit speziellem Programmierprogramm (im Fall von der SIMATIC Steuerung - SIMATIC Manager) programmiert. Wenn das Anwenderprogramm in die CPU der Steuerung übertragen wird, läuft die Steuerung im Betrieb autark ohne diesen Rechner.

Soft-SPS

Neben der vorgestellten klassischen SPS gibt es unter anderem noch die Soft-SPS (speicherprogrammierbare Steuerung in Software), deren Hardware anders konzipiert ist. Die Soft-SPS ist eine moderne Steuerung, bei der alle Funktionalitäten einer SPS vollständig in Software nachgebildet sind. Damit werden alle Hardwarekomponenten einer konventionellen SPS (SPS-Baugruppen und Programmiergerät) auf einem Rechner (zum Beispiel *Industrie-PC*) vereinigt. Es gibt zwei wichtige Punkte, die das Konzept der Soft-SPS vorangetrieben haben:

- Einen direkten Einfluss auf das Konzept hatte die Absicht zur Kostensenkung bei der Herstellung von Steuerungen. Durch Soft-SPS wird der Aufwand an Hardware deutlich verringert, denn ein Programmiergerät wird nicht mehr gebraucht. Durch die höhere Leistungsfähigkeit eines Soft-SPS ausführenden PCs kann eine SPS mehr Sensoren und Aktoren in einer Steuerung ansprechen.

- Indirekt wurde die Soft-SPS durch das hardwareunabhängige Konzept der Steuerung aus der Norm IEC 61131 (siehe Abschnitt 2.3) beeinflusst.

Bei einer klassischen SPS sind je nach CPU die Anzahl der Bausteine, die Anzahl der Datenblöcke, die Anzahl der Kommunikationsverbindungen etc. beschränkt. Ein großes Problem in der Praxis ist, dass die ausgewählte SPS (SPS ist halt teurer als ein PC) meist immer ein wenig „eng" ist und der Programmierer sich ein wenig

anstrengen muss, alles unterzukriegen, denn die nächst fähigere SPS ist auch deutlich teurer.

Gedacht ist, Soft-SPS auf gängigen Betriebssystemen wie Windows XP einzusetzen. Auf dem PC, auf dem die Soft-SPS läuft, wird ein Echtzeitbetriebssystem installiert. Windows läuft dann ebenfalls auf diesem Echtzeitbetriebssystem statt direkt auf der PC-Hardware, ohne es zu merken. Somit kann die SPS ihre Echtzeitanforderungen unabhängig von Windows erfüllen. Ein Problem für die Verlässlichkeit und Sicherheit der Steuerung sind Fehler im Rechner wie Spannungsausfall oder Speicherfehler.

2.2 Arbeitsweise

Um ein Prozess zu steuern, müssen bestimmte Informationen aus dem Prozess der Steuerung übermittelt werden. Diese Informationen werden durch Sensoren im Prozess erfasst. Die Sensoren erzeugen dann Signale, die an die Eingabebaugruppen der Steuerung übertragen werden. Die erfassten Signale gelangen als Prozessabbild der Eingabedaten (PAE) in den Arbeitsspeicher der Zentraleinheit, die für die Abarbeitung der Programme zuständig ist. Nach der Programmbearbeitung geben die Ausgabebaugruppen die von der Steuerung erstellten Signale über Prozessabbildausgaben (PAA) an den Prozess weiter. Diese Signale werden dann durch Aktoren in den Prozess eingebracht. SPS-Programme laufen zyklisch. Die Prozessabbilder werden für die Dauer des Programmlaufes stabil gehalten. Abgleich der Prozessabbilder mit den Sensoren und Aktoren findet zwischen den Programmzyklen statt und wird vom SPS-System verwaltet.

Nach dem Einschalten der SPS wird zunächst der Systemtest durchgeführt. Diese Aktion ist für den Anwender nicht sichtbar. Üblicherweise gehören zum Systemtest Speicherüberprüfung, Kontrolle der Hardwarekonfiguration, Vollständigkeitsüberprüfung der Programmteile sowie Kommunikationstest mit den E/A-Baugruppen ([Asp05]). Nach der erfolgreichen Ausführung aller Tests beginnt die SPS mit der zyklischen Bearbeitung.

Ein klassischer SPS-Zyklus wird in Abbildung 2.2 dargestellt. Am Anfang des Zyklus werden einige SPS-interne Aufgaben durchgeführt. Danach werden über die Eingänge Daten aus dem Prozess geholt und im Prozessabbild abgelegt. Nachdem die Eingänge für das Anwenderprogramm zur Verfügung stehen, wird das Anwender-

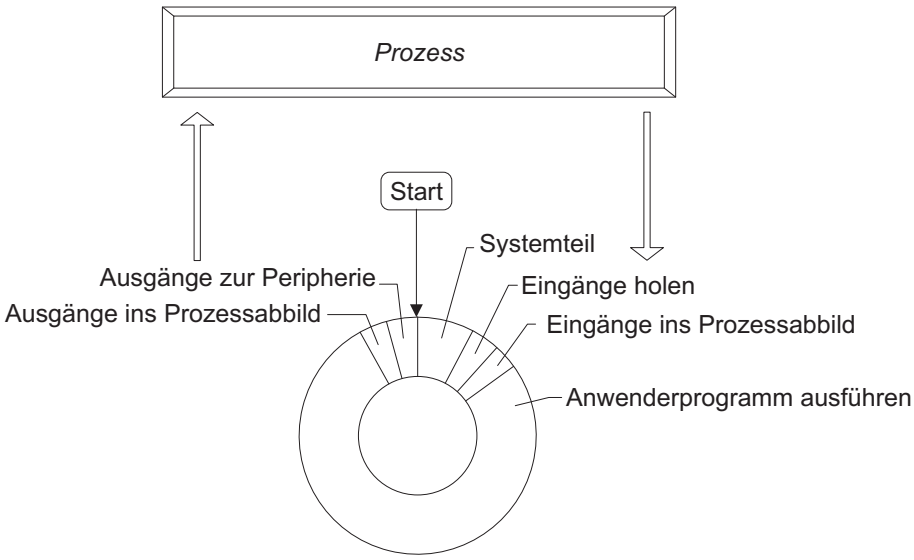


Abbildung 2.2: SPS-Zyklus

programm ausgeführt. Dies hat den größten zeitlichen Aufwand im Zyklus. Danach werden die Ausgaben im Prozessabbild gespeichert und anschließend an den Prozess weitergeliefert. Die Dauer eines Zyklus ist größtenteils von der Länge des Anwenderprogramms, also von der Ausführungszeit des Programms, abhängig. Anschließend wiederholt sich der Prozess zyklisch.

2.3 Programmierung

Die Internationale Elektronische Kommission oder abgekürzt IEC (International Electrotechnical Commission) ist eine internationale Organisation, die sich mit der Vorbereitung und Veröffentlichung von Normen im Bereich von Elektrotechnologien (Bereich von Elektrotechnik, Elektronik und anverwandten Technologien) beschäftigt. In einer der entwickelten Normen, IEC 61131-3 ([Int03]), wird unter anderem die Thematik von SPS-Programmiersprachen behandelt. Diese Sprachen werden im Folgenden kurz vorgestellt.

2.3.1 Programmiersprachen

In der Norm werden fünf Sprachen für SPS-Programmierung vorgeschlagen. Zwei davon sind textbasiert: AWL (Anweisungsliste) und ST (Strukturierter Text). Die anderen drei Sprachen sind grafische Programmiersprachen: FBS (Funktionsbaustein-

Sprache), KOP (Kontaktplan) und AS (Ablaufsprache). Dabei wird AS zur Strukturierung der internen Organisation von SPS-Programmen benutzt.

Wichtig bei den SPS-Programmiersprachen ist, dass die Norm eingehalten wird. Allerdings können die Programmiersprachen vom Hersteller anders genannt werden. Ein Beispiel dafür ist die Sprache FBS, die vom Hersteller der SIMATIC Steuerung als FUP (Funktionsplan) bezeichnet wird (siehe Abschnitt 2.4). Diese grafische SPS-Sprache wird in der vorliegenden Arbeit näher betrachtet, in dem ihre Verifikationsmöglichkeiten untersucht werden. Dabei wird ihre Syntax und Semantik detailliert beschrieben. Um dieses Kapitel einheitlich zu gestalten, wird im Folgenden die Sprache zusammen mit den anderen SPS-Programmiersprachen im Folgenden mit Hilfe von einem Beispiel kurz vorgestellt.

Anweisungsliste - AWL
(engl. Instruction List - IL)

Anweisungsliste ist eine Assembler-ähnliche Sprache, deren Name auf eine Folge von Anweisungen hinweist. Jede AWL-Anweisung muss in einer neuen Zeile beginnen. Eine solche Anweisung besteht aus einem Operator, nach dem meistens ein Operand folgt. Wenn man den Operator durch OP gezeichnet, kann die Semantik der meisten AWL-Anweisungen durch den folgenden Ausdruck beschrieben werden:

$$Ergebnis := Ergebnis\ OP\ Operand$$

Damit baut das Ergebnis der aktuellen Operation auf das Ergebnis der vorherigen Operation auf (siehe Beispiel 2.1).

Eine Anweisung kann einen optionalen *Modifizierer* enthalten, wie beispielsweise Logisches- *UND* mit Klammer „$AND($“. Nach so einer Anweisung muss der Operator „$)$“ folgen. Dabei wird der Ausdruck innerhalb der Klammern separat ausgewertet. Damit kann man im folgenden Beispiel eine *ODER*-Verknüpfung von drei Elementen betrachten, wobei das erste Element eine Konjunktion ist. Ein anderer Modifizierer ist die Negation N. Wenn er auf Logisches-*ODER* angewendet wird, bekommt man die Anweisung ORN. Damit wird eine *ODER*-Verknüpfung vom aktuellen Ergebnis und vom negierten Operand gemacht.

Die erwähnten AWL-Eigenschaften können durch das folgende Beispiel anschaulicher gemacht werden.

Beispiel 2.1. *(AWL-Beispiel)*
Durch die erste Anweisung im Beispiel wird das aktuelle Ergebnis mit dem Wert

			$OR($	
			LD	$In1$
LD	$In1$		AND	$In2$
AND	$In2$	$\Leftrightarrow$	)	
OR	$In3$		OR	$In3$
ORN	$In4$		ORN	$In4$
ST	Out		ST	Out

des Operanden In1 belegt. Danach wird das aktuelle Ergebnis mit In2 logisch UND-verknüpft. Das Alles wird mit In3 und danach mit der Negation von In4 logisch ODER-verknüpft. Anschließend wird das aktuelle Ergebnis im Operanden Out gespeichert.

Strukturierter Text - ST
(engl. Structured Text - ST)

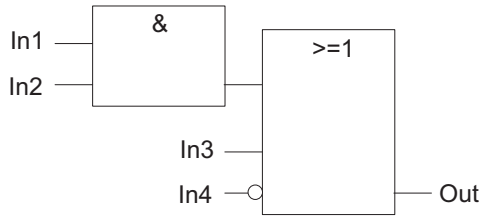
ST ist eine Pascal-ähnliche Sprache. Das heißt, dass bei ST beispielsweise folgende Anweisungen: *if-then-else*, *case*, *for*, *while* und *repeat* erlaubt sind.

Das im vorherigen Abschnitt eingeführte AWL-Beispiel wird in ST einfach durch die folgende Anweisung dargestellt:

$$Out := (In1\ and\ In2)\ or\ In3\ or\ not\ In4$$

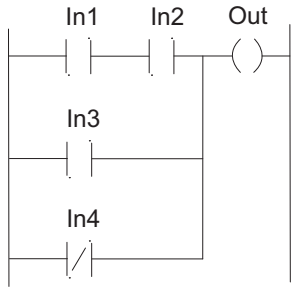
Funktionsbaustein-Sprache - FBS
(engl. Function Block Diagram - FBD)

In FBS werden Funktionsbausteine benutzt, um Operatoren darzustellen. Elemente der Sprache müssen durch Signalfluss-Linien verbunden werden. Beispiel 2.1 wird dann in FBS folgendermaßen repräsentiert:



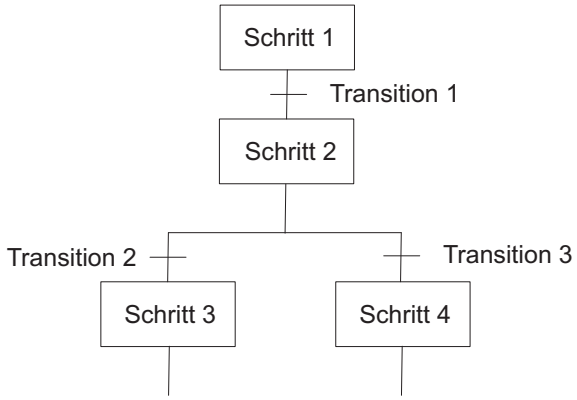
Kontaktplan - KOP
(engl. Ladder Diagram - LD)

Allgemein wird in einem SPS-Programm eine maximale Menge von grafisch miteinander verknüpften Elementen als Netzwerk bezeichnet. Bei KOP wird *Stromfluss* analog zum Fluss von elektrischem Strom in einem elektromechanischen Relais-System verwendet. In einem KOP-Netzwerk fließt der Stromfluss immer von links nach rechts. Ein Netzwerk muss dabei links durch eine vertikale Linie begrenzt werden. Diese Linie ist als *linke Stromschiene* bekannt. Analog befindet sich von der rechte Seite eine *rechte Stromschiene*. Das Beispiel 2.1 lässt sich dann in KOP folgendermaßen darstellen:



Ablaufsprache - AS
(engl. Sequential Function Chart - SFC)

AS wird benutzt, um der Steuerungsablauf zu organisieren. Die Sprache ist besonders dort geeignet, wo man im zu steuernden Prozess klare getrennte Steuerungsschritte mit zugeordneten Aktionen definieren kann. Elemente dieser Sprache sind Schritte und Transitionen, die miteinander verbunden sind. Jedem Schritt wird eine Menge von Aktionen zugeordnet. Jeder Transition wird eine Transitionsbedingung zugeordnet, die eine Weiterschaltungbedingung eines Steuerungsschrittes darstellt. Die grafische Darstellung dieser Elemente sieht folgendermaßen aus:



2.3.2 Programmiersprachen im Vergleich

AWL gehört zu den ursprünglichen SPS-Sprachen und wird von allen Herstellern zur Verfügung gestellt. Diese Assembler-ähnliche Sprache benötigt den geringsten Speicherplatz. Da AWL prozessornah ist, werden in anderen Sprachen geschriebene Programme vor dem Laden in die CPU in AWL übersetzt ([Gie05]). Die Übersetzung in anderer Richtung, also von AWL in andere Sprachen, ist nicht durchgängig möglich. FUP und KOP werden übrigens in AWL gespeichert.

Mit der zweiten Textsprache ST lässt sich SPS-Software in einer Hochsprache entwickeln. Im Vergleich zu AWL hat ST folgende Vorteile ([Hoc06]), die allerdings charakteristische Merkmale jeder Hochsprache sind:

- bessere Lesbarkeit und damit bessere Wartbarkeit der Programme,
- Programme werden schneller entwickelt,
- strukturiertes Programmieren und
- eigene Definition komplexer Datentypen.

ST-Programme haben nach der Kompilierung einen erhöhten Speicherbedarf. Dies kann bei kleineren SPS zu Problemen führen, da dann Speichergrenze schnell erreicht werden. Im Grunde gibt es zwei Strömungen bei SPS-Programmierung. Von Programmierern mit kleinen Aufgabenbereichen wird AWL bevorzugt. Programmierer für größere Anlagen verwenden die höheren Sprachen und nur zur Not AWL.

AS-Programme sind vor allem bei Großanlagen verbreitet. Die Sprache wird nicht von jedem Hersteller als Programmiersoftware angeboten. Wie bereits erwähnt,

wird AS zur Organisation der Steuerungsablauf benutzt. Allgemein können in allen Sprachen Funktionen verwendet werden, die in einer anderen SPS-Sprache geschrieben wurde. Diese Vernetzung von Sprachen ist besonders ausgeprägt in einem AS-Programm, wo jede Transition oder Aktion in einer der restlichen Sprachen geschrieben wird.

Neben AWL zählen auch KOP und FBS zu den ursprünglichen SPS-Sprachen. Als grafische Sprachen haben diese Sprachen den Vorteil, dass die Visualisierung der Programmlogik bei denen relativ leicht nachvollziehbar ist. KOP wird vor allem im amerikanischen und FBS vor allem im europäischen Einzugsbereich genutzt. Diese zwei Sprachen sind bei der SIMATIC S7 Steuerung (siehe Abschnitt 2.4) alternativ und daher in der Darstellung umschaltbar. Für die Sprachen werden bei der Steuerung Untermenge der Sprachen definiert, in denen fehlersichere Steuerungen programmiert werden können. Mehr darüber findet man im folgenden Abschnitt.

2.4 SIMATIC S7

SIMATIC S7 ist eine SPS-Familie der Firma Siemens (siehe [Gie05]). Zwei typische Vertreter sind in Abbildung 2.3 dargestellt. Das begleitende Software-Paket für die SIMATIC S7 Steuerungen heißt STEP 7. Ein Teil dieses Pakets ist der SIMATIC-Manager, der für die Projektierung von Anwendungen eingesetzt wird. Mit seiner Hilfe kann ein Projekt aufgebaut werden, indem zuerst die Hardware projektiert wird und danach die Software entwickelt wird.

Die SPS-Programmiersprachen, die in der Norm IEC 61131-3 vorgeschlagen sind, werden alle von STEP 7 angeboten. Allerdings werden dabei drei von den fünf Sprachen anders als in der Norm bezeichnet. FBS wird in STEP 7 als FUP (Funktionsplan) bezeichnet. Der in der Norm definierten textuellen Hochsprache ST entspricht S7-SCL (Structured Control Language). Eine grafische Programmiermöglichkeit für Ablaufsteuerungen wird bei STEP 7 durch S7-GRAPH ermöglicht. Ein Überblick über diese Namen wird in Tabelle 2.1 gegeben. Dort sind die STEP 7-Namen zusammen mit den Bezeichnungen aus der Norm dargestellt.

2.4.1 Sicherheitsrelevante Steuerung

Fehlersicherheit (engl. *fail-safe*) ist die Eigenschaft eines Systems, beim Auftreten eines Ausfalls im sicheren Zustand zu bleiben oder unmittelbar in einen anderen

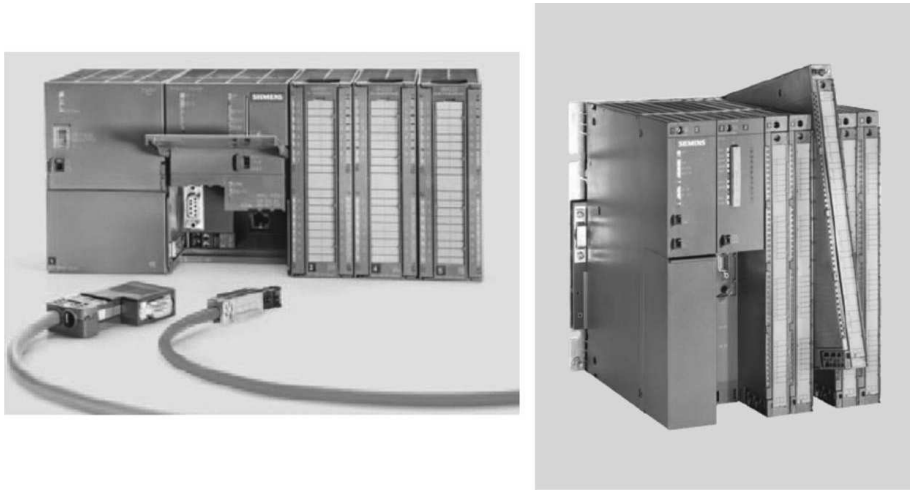


Abbildung 2.3: SIMATIC S7-300 (links) und SIMATIC S7-400 (rechts) ([Sie09])

IEC 61131-3		STEP 7	
AWL	- Anweisungsliste	AWL	
ST	- Strukturierter Text	S7-SCL	- Structured Control Language
FBS	- Funktionsbaustein-Sprache	FUP	- Funktionsplan
KOP	- Kontaktplan	KOP	
AS	- Ablaufsprache	S7-GRAPH	

Tabelle 2.1: SPS-Programmiersprachen in der Norm und in STEP 7

sicheren Zustand zu wechseln.

Der Hersteller von SIMATIC bietet eine Automatisierungsplattform zur Integration sicherheitstechnischer Funktionen in ihren Standardprodukten (siehe [Sie09]). Das Konzept wird *safety integrated* genannt. Bei SIMATIC Safety Integrated Steuerungen steht die umfassende Funktionalität von SIMATIC zur Verfügung. Zusätzlich haben diese Steuerungen die Eigenschaften sich selbstständig zu überwachen, selbstständig Fehler zu erkennen und unmittelbar beim Auftreten eines Fehlers in einen sicheren Zustand zu wechseln. Als Beispiel kann man die Steuerungen S7-300F und S7-400F erwähnen. Hier wird „F" benutzt, um die Fehlersicherheit der Steuerung zu bezeichnen. Analog wird diese Bezeichnung bei anderen Begriffen im Laufe des Abschnitts benutzt.

Für die Konfiguration und Programmierung von fehlersicheren Systemen wird eine Baustein-Bibliothek mit zertifizierten Beispielen benutzt, die im Software-Paket *S7 Distributed Safety* zur Verfügung gestellt wird. Für die Programmierung wird

dann weiter der Standard-STEP 7 Editor für Sprachen FUP und KOP benutzt. Programmiert wird allerdings in F-FUP und F-KOP. Der Unterschied zwischen F-FUP/F-KOP und entsprechenden Standard-Sprachen liegt hauptsächlich in der Menge der Anweisungen, den benutzten Datentypen und den Adressenbereichen. Zum Beispiel sind in F-FUP/F-KOP Datentypen wie *REAL*, *STRING* und *ARRAY* nicht erlaubt.

Von großer Bedeutung bei sicherheitsrelevanten Systemen ist, die Korrektheit einer Anwendungsapplikation für das System zu gewährleisten. Ein Beitrag dazu liefert die vorliegende Arbeit, in der ein Verfahren für die Verifikation von FUP-Software vorgestellt wird. Das Verifikationsverfahren lässt sich auch auf F-FUP anwenden. Zu den Systemen mit Sicherheitsverantwortung gehören unter anderem Systeme aus der Domain der Eisenbahnautomatisierung. Eine Softwarekomponente aus dieser Domain wird als Fallstudie in dieser Arbeit benutzt.

2.4.2 Einblick ins Arbeitsregister

In der vorliegenden Arbeit wird ein Verfahren für die FUP-Verifikation vorgestellt. Um ein FUP-Programm mit diesem Verfahren zu verifizieren, werden verschiedene Transformationen auf Basis des AWL-Formats des Programms durchgeführt. Im ersten Schritt dieser Transformationskette wird von Bedeutung sein, wie die Ergebnisbildung einer AWL-Anweisung abläuft. Allgemein hängt das von den hardwarespezifischen Merkmalen der konkreten Steuerung ab.

In der Norm IEC 61131-3 wird ein hardwareunabhängiger Ansatz für die SPS-Programmierung vorgestellt. Da in dieser Arbeit die SIMATIC S7-Steuerung benutzt wird, werden manche Hardwaremerkmale dieser Steuerung im Folgenden kurz eingeführt. Ein einfaches Konzept enthält folgende Elemente (siehe Abbildung 2.4):

- Akkumulatoren - Abhängig vom SPS-System gibt es bei SIMATIC S7 zwei oder vier Akkumulatoren mit der Breite von 32 Bit, die auch Akkus genannt werden. Die Akkus werden als Stack organisiert angesprochen. AWL ist eine ein Operanden Maschine, die entweder die oberen Stackwerte verarbeitet oder einen externen Wert mit einem Stackwert verknüpft. Bei der Ergebnisbildung einer Anweisungen wird auf diese Akkumulatoren zugegriffen. Beispielsweise wird mit der Anweisung $CMP{==}I$ überprüft, ob $Akku1$ den gleichen Inhalt wie $Akku2$ hat.

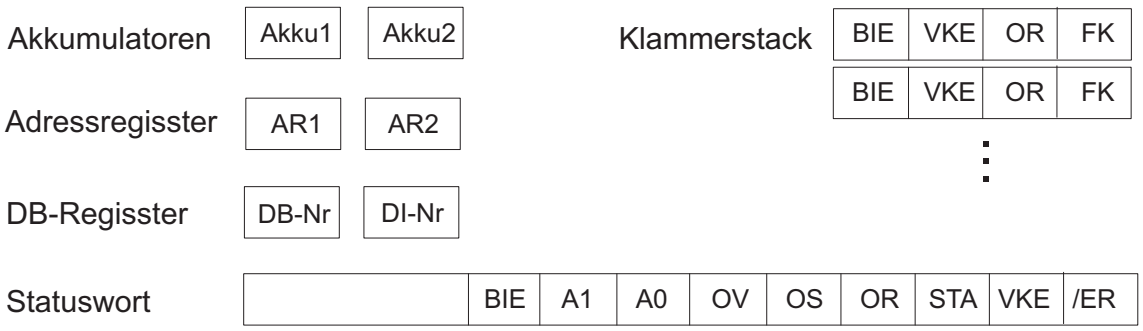


Abbildung 2.4: Register der SIMATIC S7

- Adressregister - Für die indirekte Adressierung stehen zwei 32 Bit Register zur Verfügung.

- DB-Register - Es gibt zwei DB-Register, die den Verweis auf die aktuell geöffneten Datenbausteine (DB) enthalten. In einem wird der Verweis auf den globalen DB und im anderen auf den Instanz-DB aufbewahrt.

- Statuswort - Im Statuswort werden folgende Status-Bits zusammengefasst: Binärerergebnis (BIE), Anzeige 1 (A1), Anzeige 2 (A2), Überlauf (OV), Überlauf speichernd (OS), OR-Bit (OR), Statusbit (STA), Verknüpfungsergebnis (VKE) und Erstabfrage (/ER). Besonders wichtig für spätere Betrachtungen ist das VKE-Bit, das dem aktuellen Ergebnis aus Abschnitt 2.3.1 entspricht. Auf die Status-Bits wird hier nicht weiter eingegangen.

  An dieser Stelle kann noch ein Vergleich zwischen IEC-AWL und STEP 7-AWL gemacht werden, indem Beispiel 2.1 in STEP7 dargestellt wird: Also,

$$\begin{array}{ll} U & In1 \\ U & In2 \\ O & In3 \\ ON & In4 \\ = & Out \end{array}$$

  mit der ersten Anweisung wird der Wert von $In1$ in VKE gespeichert. Danach wird VKE mit $In2$ konjugiert. Das Ergebnis wird mit $In3$ und danach mit der Negation von $In4$ logisch $ODER$-verknüpft. Anschließend wird VKE im Operanden Out gespeichert.

- Klammerstack - Der Klammerstack ist ein Speicherbereich, den die Verknüpfungsoperationen mit Klammer (beispielsweise „U(" oder „O(") verwenden. Wenn eine solche Anweisung vorkommt, werden bestimmte Daten auf den Stack gespeichert. Dann wird eine neue Bitverknüpfungsoperation gestartet, die mit der Anweisung „)" endet. Danach werden die Daten aus dem Stack geholt und weiterbearbeitet.

3 Formale Verifikation

Der Begriff *formale Methoden* verweißt auf Anwendung von Techniken aus Logik und diskreter Mathematik in Spezifikation, Design und Entwurf von Computersystemen und Software ([NAS97]). Das Ziel bei Anwendung formaler Methoden ist, Entscheidungen ohne menschliche Intuition und anhand formaler Regeln zu treffen. Damit steht die Entwicklung formaler Methoden in einer engen Verbindung mit der Entwicklung künstlicher Intelligenz und mit der Idee, ein Programm für die Simulation menschlichen Denkens zu entwickeln.

Es gibt mehrere Verfahren, mit denen man überprüfen kann, ob ein System gewünschte Eigenschaften erfüllt oder nicht: Simulation, Test oder Verifikation. Beim Testen wird das Verhalten eines Systems bei Ausführung eines Versuchs beobachtet. Im Gegensatz zum Testen wird bei der Simulation das Verhalten einer Abstraktion (eines Modells) des Systems beobachtet. Sowohl mit der Simulation als auch mit dem Testen kann nur die Anwesenheit von Fehlern, jedoch nicht ihre Abwesenheit gezeigt werden. Mit beiden Methoden wird überprüft, ob das System in einem konkreten Zustand eine gewünschte Eigenschaft erfüllt oder nicht. In einem positiven Fall heißt das noch nicht, dass die Eigenschaft im ganzen System erfüllt wird. Um das festzustellen, sollte man für jeden möglichen Zustand den selben Versuch ausführen. Für große Systeme wird so etwas aus Kosten- und zeitlichen Gründen zu aufwändig oder sogar unmöglich.

Im Unterschied zur Simulation und zum Testen wird mit Verifikation auch die Abwesenheit von Fehlern gezeigt. Verifikationsverfahren können deduktiv (regelbasiert) oder modellbasiert sein. Die weit verbreitete Begriffe für automatisierte deduktive bzw. modellbasierte Verifikation sind Theorem Proving bzw. Model Checking. Bevor eine genauere Beschreibung der beiden Verfahren in den folgenden Unterkapiteln gegeben wird, werden einige grundlegende Begriffe aus der Theorie formaler Methoden im nächsten Kapitel kurz erklärt.

3.1 Grundlagen

Im Allgemeinen soll bei formaler Verifikation überprüft werden, ob ein System gewünschte Eigenschaften besitzt. Bei Rechnersystemen werden dabei Hardware und Software unabhängig voneinander betrachtet. Die ersten beiden Schritte bei der Verifikation sind das System zu modellieren und die Systemspezifikation in Form logischer Formeln darzustellen. Lösungen dafür werden in diesem Abschnitt vorgestellt. Der dritte Schritt ist die Erfüllbarkeit einer Formel im Modell zu überprüfen. Dies wird in Abschnitt 3.2 (Theorem Proving) und Abschnitt 3.3 (Model Checking) behandelt.

Wie bereits gesagt befasst sich dieser Abschnitt mit der Systemmodellierung und den Logiken zur Spezifikationsbeschreibung. Dabei unterscheiden sich Prinzipien deduktiver und modellbasierter Verifikation sehr. Da der Schwerpunkt der vorliegenden Arbeit im Model Checking liegt, werden hier Modelle zur Systembeschreibung und Logiken zur Spezifikationsbeschreibung hauptsächlich aus der Sicht des Model Checking näher erklärt.

3.1.1 Modelle zur Systembeschreibung

Es gibt viele Strukturen für die Modellierung eines Systems, die geeignet für die Anwendung formaler Verifikation sind: Kripke-Strukturen, Transitionssysteme, Ereignissysteme, Petrinetzen, usw. Die ersten zwei Strukturen werden hier näher betrachtet.

Kripke-Strukturen

Kripke-Strukturen wurden bereits im Jahr 1963 von S.A. Kripke eingeführt ([Kri63a, Kri63b]). Ihr grundlegendes Modell kann folgendermaßen definiert werden.

Definition 3.1. *(Kripke-Struktur)*
Sei P eine Menge der atomaren Aussagen. Ein Tupel $K = (Q, Q_0, \delta, l)$ ist eine Kripke-Struktur über P, wenn seine Komponenten wie folgt definiert sind:

- *Q ist eine Menge der Zustände,*
- *Q_0 ist eine Menge der Startzustände,*
- *$\delta \subset Q \times Q$ ist eine Zustandsübergangsrelation,*

- $l : Q \rightarrow 2^P$ *ist eine Beschriftungsfunktion (Labelfunktion).*

Ein System kann mit einer Kripke-Struktur in Form von Zuständen und Zustandsübergängen definiert werden. Mit Hilfe der Labelfunktion werden Zustände mit der Menge der Eigenschaften (atomaren Aussagen) markiert. Systemeigenschaften können als Belegungen von Systemvariablen betrachtet werden. In dem Sinne enthält die Beschriftung jedes Zustands die Information, mit welchem Wert jede Variable in dem Zustand belegt ist.

Transitionssysteme

Neben Kripke-Strukturen werden auch Transitionssysteme für Systemmodellierung benutzt. Das Konzept eines Transitionssystems ähnelt dem Konzept einer Kripke-Struktur. Die Konzepte haben allerdings verschiedene Schwerpunkte. Der Schwerpunkt von Kripke-Strukturen liegt in den Zuständen. Im Gegensatz dazu liegt der Schwerpunkt von Transitionssystemen in den Transitionen, die durch Aktionen gekennzeichnet werden. Diese Modelle werden für die Modellierung von Systemen verwendet, deren Zustände durch Aktionsausführungen bestimmt sind.

Definition 3.2. *(Transitionssystem)*
Seien P eine Menge der atomaren Aussagen und A eine Menge der Aktionen. Ein Tupel $T = (Q, Q_0, \delta, l)$ ist ein Transitionssystem über atomaren Aussagen P und Aktionen A, wenn seine Komponenten Folgendes erfüllen:

- Q *ist eine Menge der Zustände,*
- Q_0 *ist eine Menge der Startzustände,*
- $\delta \subset Q \times A \times Q$ *ist eine Zustandsübergangsrelation,*
- $l : Q \rightarrow 2^P$ *ist eine Beschriftungsfunktion (Labelfunktion).*

Andere Modelle

Es gibt viele andere Strukturen, die für die Systemmodellierung benutzt werden können. Einen Überblick darüber bietet Tabelle 3.3, die eine tabellarische Darstellung der untersuchten Veröffentlichungen aus Abschnitt 3.4 repräsentiert.

3.1.2 Temporale Logiken zur Spezifikationsbeschreibung

Eine Systemspezifikation wird oft mithilfe einer temporalen Logiken dargestellt. Dabei können Aussagen formuliert werden, die Systemzustände zeitlich verbinden. Beispiele solcher Aussagen sind: „Irgendwann in der Zukunft erfüllt das System S die Eigenschaft φ“ oder „Die Eigenschaft φ wird im System S immer erfüllt“. Es gibt verschiedene Interpretationen von temporalen Logiken. Das Konzept wird detaillierter in den folgenden Abschnitten vorgestellt, indem Linear Temporal Logic (LTL) und Computation Tree Logic (CTL) eingeführt werden.

Linear Temporal Logic (LTL)

Mit LTL-Formeln können Eigenschaften formuliert werden, die auf einer sequentielen Systemausführung basieren. Das heißt, es wird immer ein Berechnungspfad des Systems betrachtet. Wenn es verschiedene Lebenszyklen des Systems gibt, findet eine implizite Allquantifizierung statt und es werden alle Pfade berücksichtigt. LTL-Syntax kann durch folgende Definition beschrieben werden.

Definition 3.3. *(Syntax von LTL)*
Sei P eine Menge der atomaren Aussagen. Die Menge der LTL-Formel über P wird in BNF folgendermaßen definiert:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid X\varphi \mid \varphi_1 \, U \, \varphi_2$$

wobei $p \in P$.

In LTL-Formeln werden auch weitere Operatoren benutzt, die durch Operatoren aus der Definition definiert werden, wie zum Beispiel:

- $\bot \equiv \neg\top$,
- $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$
- $F\varphi \equiv \top \, U \, \varphi$
- $G\varphi \equiv \neg F \neg\varphi \equiv \neg(\top \, U \, \neg\varphi)$.

Eine intuitive Semantik von LTL temporalen Operatoren wird in Abbildung 3.1 erfasst.

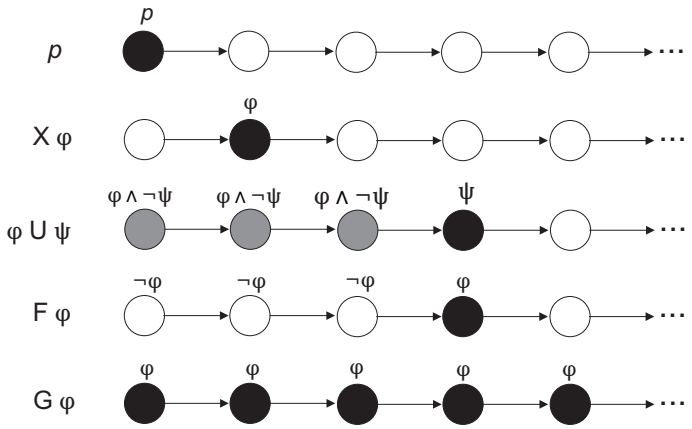


Abbildung 3.1: Intuitive Semantik der LTL

Computation Tree Logic (CTL)

Bei CTL wird das Verhalten eines Systems in Form eines Berechnungsbaums betrachtet. Die Wurzel des Baums entspricht dem Startzustand. Die Nachfolger eines Elements im Baum werden im Bezug zum entsprechenden Übergang im Systemmodell bestimmt. Im Gegensatz zur LTL bezieht sich die Interpretation der CTL auf verschiedene Berechnungspfade. Damit können in einer CTL-Formel verschiedene Lebenszyklen eines Systems berücksichtigt werden. Mit Pfadquantoren E bzw. A kann man Eigenschaften definieren, die sich auf einen Pfad bzw. alle Pfade im Berechnungsbaum beziehen.

Definition 3.4. *(Syntax von CTL)*
Sei P eine Menge der atomaren Aussagen. Die Menge der CTL-Formel über P wird in BNF wie folgt definiert:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid EX\varphi \mid E\,[\varphi_1\, U\, \varphi_2] \mid A\,[\varphi_1\, U\, \varphi_2]$$

wobei $p \in P$.

Mithilfe der in der Definition aufgeführten CTL-Operatoren können auch weitere Operatoren definiert werden, wie zum Beispiel:

- $EF\varphi \equiv E\,[\top\, U\, \varphi]$
- $AG\varphi \equiv \neg EF(\neg\varphi) \equiv \neg E\,[\top\, U\, \neg\varphi]$

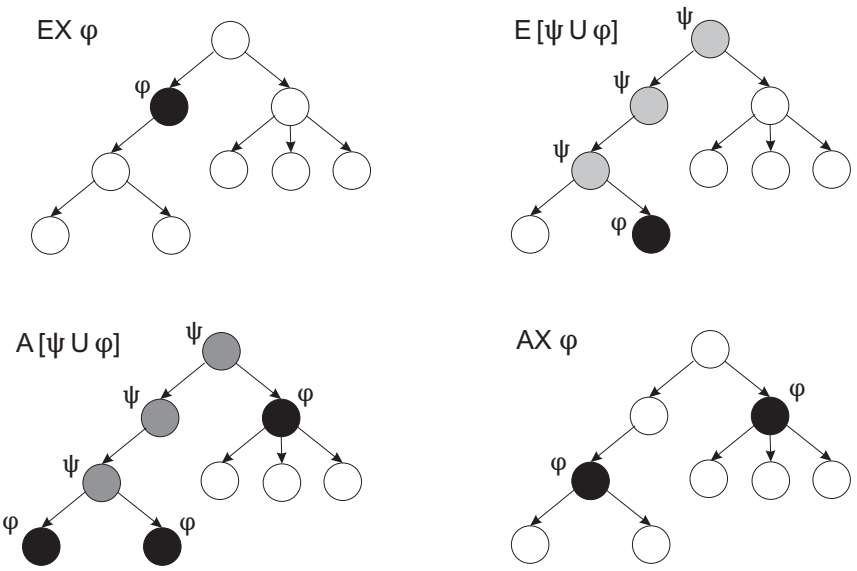


Abbildung 3.2: Intuitive Semantik der CTL

3.2 Theorem Proving

Theorem Proving oder automatische Deduktion bezieht sich auf den Mechanismus des deduktiven logischen Denkens. Im Gegensatz zum Model Checking lassen sich deduktive Methoden auf unendliche Systeme anwenden. Um die Gültigkeit einer Eigenschaft zu zeigen, wird bei deduktiver Verifikation ein Beweis anhand der definierten Axiome und Beweisregeln geführt. Anfangs wurde diese Technik per Hand durchgeführt. Sie hat eine Anwendung bei sicherheitskritischen Systemen gefunden, bei denen es wichtig war, eine Eigenschaft zu beweisen und der zeitliche Aspekt nebensächlich war.

Obwohl es mittlerweile auch automatische Theorem Prover gibt, wird Beweisführung oft halbautomatisch durchgeführt. Bei der interaktiven Beweisführung soll an der einen oder anderen Stelle der Benutzer entscheiden, in welche Richtung der Beweis weitergeführt wird. Deswegen sollte der Benutzer viel Erfahrung im Logikbereich und in der Beweisführung haben. Sogar im Fall, wenn ein Experte die Verifikation durchführt, ist dies ein zeitaufwendiger Prozess.

Nachfolgend werden Grundbegriffe des Theorem Proving erklärt. Um ein besseres Bild über die Vorgehensweise dieser Technik zu bekommen, wird anschließend ein Beispiel der Beweisführung gegeben.

3.2.1 Beweistheorie

Ein *formales System* enthält folgende Elemente:

- ein Alphabet und eine nichtleere Menge der Worte darüber - *Primitiven*;
- eine Menge von Grundaussagen über Primitiven - *Axiome*; und
- eine Menge von Schlussregeln oder Herleitungen anderer Art, mit denen man neue Aussagen ableiten kann - *Theoremen*.

Axiome und Schlussregeln eines formalen Systems bilden ein *deduktives System*. Axiome zusammen mit beweisbaren und bereits bewiesenen Theoremen bilden eine *Theorie*. Ein *Beweis* eines Theorems ist eine Reihe der Transformationen, die mit Schlussregeln übereinstimmen (siehe Abschnitt 3.2.4).

Mit $\vdash_L \varphi$ bezeichnet man, dass φ in der Logik L beweisbar ist. Das heißt, dass entweder φ ein Theorem in der Logik L ist, oder φ kann man anhand von Axiomen beweisen, ohne andere Annahmen zur Hilfe zu nehmen. Wenn die Logik eindeutig ist, wird nur $\vdash \varphi$ geschrieben. Im Allgemeinen braucht man für den Beweis eines Satzes φ neben Axiomen auch zusätzliche Behauptungen $\gamma_0, \gamma_1, \ldots, \gamma_n$. Dann schreibt man $\gamma_0, \gamma_1, \ldots, \gamma_n \vdash \varphi$.

So ein syntaktisch dargestelltes, formales System wird *Kalkül* genannt. Die Studie über rein formale oder syntaktische Eigenschaften eines Kalküls heißt Beweistheorie.

3.2.2 Modelltheorie

Die Unterscheidung zwischen Syntax und Semantik in der Logik spiegelt sich in der Unterscheidung zwischen formalen Systemen und ihren Interpretationen wider. Beim Theorem Proving wird dabei über Beweistheorien und Modelltheorien gesprochen. Beispielsweise sind natürliche Zahlen ein Modell der Peano-Axiome. Ein System kann auch zwei verschiedene Interpretationen haben. Zum Beispiel ist Planimetrie ein Model der Euklidischen Axiome und ein anderes Modell dafür ist hyperbolische Geometrie.

In einer Interpretation kann jedem syntaktisch korrekten Satz aus einem Kalkül ein boolescher Wert zugewiesen werden. Der Wert wird *true* oder *false* sein, abhängig davon, ob der Satz in der Interpretation richtig oder falsch ist. Eine Interpretation ist ein Modell eines formalen Systems, wenn alle Axiome in der Interpretation richtig

sind. Ähnlich ist eine Interpretation ein Modell einer Theorie, wenn alle Sätze in der Theorie als richtig bewertet sind.

Sei φ ein Satz im Kalkül und I eine Menge der Interpretationen des Kalküls. φ ist allgemeingültig ($\models \varphi$), genau dann wenn φ in jeder Interpretation aus I richtig ist. Wenn jedes Modell einer Menge der Aussagen S, auch ein Modell der Aussage φ ist, wird geschrieben $S \models \varphi$ (S enthält φ).

3.2.3 Sequenzenkalkül als Beispielkalkül

Es gibt im Allgemeinen zwei Arten von Herleitungen ([BB89]). Beim deduktiven Kalkül gehen Schlussregeln von Axiomen aus, bis die gewünschte Formel bewiesen wird. Andersrum funktionieren Schlussregeln beim Testkalkül. Dabei wird von der Formel ausgegangen, bis die Axiome hergeleitet worden sind. In diesem Abschnitt wird eine Variante des deduktiven Kalküls näher erläutert - Sequenzenkalkül, der nach seinem Entwickler G. Gentzen auch Gentzenkalkül genannt wird.

Eine Sequenz beim Sequenzenkalkül wird durch $\Gamma \vdash \Delta$ bezeichnet, wobei Γ eine Konjunktion, $A_1 \wedge \ldots \wedge A_m$, und Δ eine Disjunktion, $B_1 \vee \ldots \vee B_n$, logischer Formeln ist. Die Formeln in Γ werden Antezedenz und die Formeln in Δ Konsequenz genannt. Intuitiv heißt $\Gamma \vdash \Delta$, dass aus der Antezedenz die Konsequenz folgt ($A_1 \wedge \ldots \wedge A_m \supset B_1 \vee \ldots \vee B_n$).

Einige Schlussregeln im Sequenzenkalkül sind:

- „präp-axiom“

$$\frac{}{A, \Gamma \vdash A, \Delta}\ Ax$$

  Diese Schlussregel des präpositionalen Axioms wird angewendet, wenn eine Formel sowohl in der Antezedenz als auch in der Konsequenz erscheint ($A \wedge \Gamma \supset A \vee \Delta$).

- „neg-links“ und „neg-rechts“

$$\frac{\Gamma \vdash A, \Delta}{\neg A, \Gamma \vdash \Delta}\ \neg \vdash \qquad\qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\ \vdash \neg$$

  Diese beiden Negationsregeln bedeuten, dass die Negation einer Formel in der Antezedenz äquivalent dem Erscheinen der gleichen Formel in der Konsequenz ist, und umgekehrt. Das lässt sich anhand der Identität $(X \supset Y) \equiv (\neg X \vee Y)$ und der De Morgans Regel $\neg(X \vee Y) \equiv (\neg X \wedge \neg Y)$ zeigen. Beispielsweise kann

man die erste Regel folgendermaßen ableiten:

$$(\neg A \wedge \Gamma) \supset \Delta \;\equiv\; \neg(\neg A \wedge \Gamma) \vee \Delta \;\equiv\; (A \vee \neg\Gamma) \vee \Delta \;\equiv$$
$$\neg\Gamma \vee (A \vee \Delta) \;\equiv\; \Gamma \supset (A \vee \Delta)$$

- „impl-rechts“

$$\frac{A, \Gamma \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta} \vdash\!\to$$

Die Regel für die Implikation auf der rechten Seite kann man mit Hilfe der linken Negation folgendermaßen ableiten:

$$\Gamma \supset (A \to B) \vee \Delta \;\equiv\; \Gamma \supset (\neg A \vee B) \vee \Delta \;\equiv$$
$$\Gamma \supset \neg A \vee (B \vee \Delta) \;\equiv\; A \wedge \Gamma \supset B \vee \Delta \quad .$$

- „impl-links“

$$\frac{\Gamma \vdash A, \Delta \qquad B, \Gamma \vdash \Delta}{A \to B, \Gamma \vdash \Delta} \to\!\vdash$$

Bei der Implikation auf der linken Seiten wird der Beweis in zwei Fälle gespalten: $\Gamma \vdash A, \Delta$ und $B, \Gamma \vdash \Delta$. Das kann man wie folgt ableiten:

$$(A \to B) \wedge \Gamma \supset \Delta \;\equiv\; (\neg A \vee B) \wedge \Gamma \supset \Delta \;\equiv$$
$$(\neg A \wedge \Gamma \supset \Delta) \vee (B \wedge \Gamma \supset \Delta) \;\equiv\; (\Gamma \supset A \vee \Delta) \vee (B \wedge \Gamma \supset \Delta)$$

- „und-links“:

$$\frac{A, B, \Gamma \vdash \Delta}{A \wedge B, \Gamma \vdash A, \Delta} \wedge\!\vdash$$

Die Regel für die Konjunktion auf der linken Seite ist einfach die Folge der Tatsache, dass die Elemente in der Antezedenz mit $\wedge$ verbunden sind.

3.2.4 Ein Beispiel der Beweisführung

Um einen besseren Einblick ins Theorem Proving zu ermöglichen, wird an dieser Stelle ein Beispiel der Beweisführung gegeben. Es gibt viele Theorem Prover, deren Handhabung in entsprechenden Handbüchern ausführlich beschrieben wird: Isabelle ([NPW09]), PVS ([SSRSC01]), Coq ([INR09]). Das folgende Beispiel hat nicht zum Ziel, Eigenschaften eines konkreten Theorem Provers und seine Umgebung vorzustellen, sondern einfach die allgemeine Denkweise beim Theorem Proving zu demonstrieren. Als Quelle für das Beispiel wird [NAS97] genommen, in der man eine detaillierte Beschreibung der Grundlagen formaler Methoden finden kann.

Beispiel 3.1. *$((P \supset Q \supset R) \supset (P \wedge Q) \supset R)$*
In diesem Beispiel soll gezeigt werden, wie der Beweis für die Formel $(P => (Q =>$

R)) => ((P & Q) => R) abgeleitet werden kann. In den folgenden Beweisschritten wird immer zuerst der Name der anzuwendenden Schlussregeln angegeben. Die Zeile „====>“ wird benutzt, um die Antezedenz von der Konsequenz zu trennen. Der Beweis wird in der folgende Tabelle zusammengefasst. Anschließend folgt eine kurze Erläuterung der verwendeten Regeln.

Schritt 0: *Anfang des Beweises*

Kein
====>
Formel 1: (P => (Q => R)) => ((P & Q) => R)

Schritt 1: *Verwende die Regel „impl-rechts“*

Formel 1: P => (Q => R)
====>
Formel 1: (P & Q) =>R

Schritt 2: *Verwende die Regel „impl-rechts“*

Formel 1: P => (Q => R)
Formel 2: P & Q
====>
Formel 1: R

Schritt 3: *Verwende die Regel „und-links“*

Formel 1: P => (Q => R)
Formel 2: P
Formel 3: Q
====>
Formel 1: R

Schritt 4: *Verwende die Regel „impl-links“*

Formel 1: Q => R	*Formel 1: P*
Formel 2: P	*Formel 2: Q*
Formel 3: Q	*====>*
====>	*Formel 1: P*
Formel 1: R	*Formel 2: R*

Schritt 4.1: *Verwende „impl-links“*

Formel 1: R	*Formel 1: P*
Formel 2: P	*Formel 2: Q*
Formel 3: Q	*====>*
====>	*Formel 1: Q*
Formel 1: R	*Formel 2: P*

Schritt 4.1.1: *„präp-axiom“*
Schritt 4.1.2: *„präp-axiom“*

Schritt 4.2: *Verwende „präp-axiom“*

Tabelle 3.1: Beweisführung beim Theorem Proving

Am Anfang des Beweises ist die Antezedenz leer. In jedem Schritt danach wird eine von den folgenden Regeln angewendet:

„impl-rechts“ *Diese Regel lässt sich auf eine Formel der Form $X => Y$ anwenden. Um die Richtigkeit dieser Formel zu zeigen, wird nach der Anwendung der Regel X als Antezedens betrachtet. Dann bleibt noch die Richtigkeit von Y zu zeigen.*

„und-links“ *Wenn die Formel $X \& Y$ als Antezedenz erscheint, entstehen aus der Antezedenz nach der Anwendug dieser Regel zwei Antezdenzen, X und Y.*

„impl-links“ *Wenn $X => Y$ in der Antezedenz vorkommt, kann die Regel „impl-links“ angewendet werden. Dann entstehen zwei neue Formeln, die bewiesen werden sollen (siehe die Erklärung dieser Regel im vorherigen Abschnitt):*

- *Statt $X => Y$ in der Antezedenz wird X in der Konsequenz betrachtet.*
- *Statt $X => Y$ erscheint X in der Antezedenz.*

„präp-axiom“ *Mit dieser Regel wird eine Formel bewiesen, die bereits in der Antezedenz vorkommt.*

3.3 Model Checking

Eine weitere Verifikationsmethode ist Model Checking. Mit dem Model Checking werden endliche Systeme verifiziert. Erste Versuche einer manuellen Beweisführung anhand modellbasierter Verifikation wurden schon Ende der 70er durchgeführt. In [Pnu77] schlägt Pnueli vor, wie man temporale Logiken und Schlussfolgerung bei nebenläufigen und reaktiven Programmen nutzen kann. Das erste Mal wurde das Verfahren des Model Checking unabhängig voneinander von Clarke und Emerson in [CE81] und Quielle und Sifakis in [QS82] veröffentlicht. Eine genauere Auflistung über alle diese Ideen kann in [CGL93] gefunden werden.

Beim Model Checking wird das folgende Problem betrachtet. Sei die Kripke-Struktur $M = (Q, Q_0, \delta, l)$ ein Modell eines endlichen Systems und φ eine logische Formel, die eine Eigenschaft des Systems repräsentiert, dann sollen alle Systemzustände gefunden werden, die die Formel φ erfüllen: $\{q \in Q \mid M, q \vDash \varphi\}$. Wenn dieser Menge alle initialen Zustände angehören, wird gesagt, dass das Modell M die Formel φ erfüllt.

Abhängig von der Logik, die beim Model Checking benutzt wird, kann man zwischen LTL- und CTL-Model Checking unterscheiden. Als Quelle für ihre Beschreibung wurden [CGP99, BK08, Kat99, HR00] benutzt.

3.3.1 LTL-Model Checking

Ein LTL-Model Checking Algorithmus soll entscheiden, ob ein Transitionssystem eine LTL-Formel erfüllt. Die Basis dieser Technik liegt im Ansatz der Automatentheorie, der sich auf *Büchi-Automaten* bezieht. Um das LTL-Model Checking Verfahren näher zu beschreiben, wird zunächst eine Definition des Büchi-Automaten angegeben.

Büchi-Automaten

Definition 3.5. *(Nichtdeterministischer Büchi-Automat (NBA))*
Sei $S = (\Sigma, Q, Q_0, \delta, F, l)$ ein Tupel mit folgenden Elementen:

- Σ - *ein Alphabet,*
- Q - *eine Menge der Zustände,*
- $Q_0 \subset S$ - *eine Menge der Startzustände,*
- $\delta : Q \rightarrow 2^Q$ - *eine Zustandsübergangsrelation,*
- $F \subset Q$ - *eine Menge der akzeptierten Zustände, und*
- $l : Q \rightarrow \Sigma$ - *eine Beschriftungsfunktion (Labelfunktion).*

S ist ein nichtdeterministischer Büchi-Automat (NBA), wenn S unendliche Eingabeworte über Σ erkennt.

Definition 3.6. *(Lauf über NBA)*
$\sigma = q_0\, q_1 \ldots$ ist ein Lauf über einen NBA $(\Sigma, Q, Q_0, \delta, F, l)$, wenn $q_0 \in Q_0$ und $(q_i, q_{i+1}) \in \delta, \forall i \geq 0$. Sei $lim(\sigma)$ die Menge aller Zustände, die unendlich oft in σ vorkommen. σ wird genau dann akzeptiert, wenn $lim(\sigma) \cap F \neq \varnothing$.

Bezeichnet man die Menge der unendlichen Worte über Σ mit Σ^ω, dann wird ein Wort $\omega = a_0\, a_1 \ldots \in \Sigma^\omega$ genau dann akzeptiert, wenn es einen akzeptierten Lauf $\sigma = q_0\, q_1 \ldots$ mit der Eigenschaft $l(q_i) = a_i, \forall i \geq 0$ gibt. Die Sprache eines NBA S, $\mathcal{L}_\omega\{S\}$, ist die Menge der unendlichen Worte, die von S akzeptiert werden:

$$\mathcal{L}_\omega(S) = \{\omega \in \Sigma^\omega \mid \omega \textit{ wird von } S \textit{ akzeptiert}\}.$$

Nach [Kat99] kann jede LTL-Formel φ mit einem NBA S_φ, repräsentiert werden, der alle unendlichen Worte über atomaren Aussagen akzeptiert, die die Formel φ erfüllen.

Verifikationsverfahren

Eine intuitive Lösung für das Problem LTL-Model Checking könnte in den folgenden drei Schritten beschrieben werden:

- konstruiere einen Büchi-Automat für φ, S_φ,
- konstruiere einen Automat für das System S,
- überprüfe, ob $\mathcal{L}_\omega(S) \subseteq \mathcal{L}_\omega(S_\varphi)$.

Leider gibt es im allgemeinen Fall keinen effizienten Algorithmus, der entscheiden kann, ob eine Sprache eine Teilmenge einer anderen Sprache ist.

Die Hauptidee des LTL-Model Checking ist $S \vDash \varphi$ zu beweisen, indem man nach einem Pfad π in S sucht, der $\neg\varphi$ erfüllt. Falls so ein Pfad existiert, liefert π das Gegenbeispiel für die Aussage. Wenn π nicht gefunden werden kann, heißt es $S \vDash \varphi$.

Um zu überprüfen, ob S die Bedingung $\neg\varphi$ erfüllt, werden beim LTL-Model Checking folgende drei Schritte gemacht:

1. es wird ein Büchi-Automat $S_{\neg\varphi}$ für die Formel $\neg\varphi$ konstruiert;
2. es wird ein Büchi-Automat S für das System konstruiert;
3. es wird überprüft, ob $\mathcal{L}_\omega(S) \cap \mathcal{L}_\omega(S_{\neg\varphi}) = \varnothing$.

Im Gegensatz zur Inklusion, lässt sich der Durchschnitt zweier Sprachen algorithmisch effizient beschreiben. Es wird nämlich ein Kreuzprodukt von S und S_φ konstruiert. Der Automat $S \otimes S_\varphi$ akzeptiert die Sprache $\mathcal{L}_\omega(S) \cap \mathcal{L}_\omega(S_{\neg\varphi})$. Es wird dann noch überprüft, ob $\mathcal{L}_\omega(S \otimes S_{\neg\varphi}) = \varnothing$.

Komplexität

Die Komplexität des LTL-Model Checking Verfahrens hängt von der Komplexität der Algorithmen für die Konstruktion der Automaten $S_{\neg\varphi}$ und S ab. Bei der Darstellung einer Formel durch einen Automaten wird zuerst ein Graph erstellt, dessen Knoten durch eine Menge der Teilformeln von φ gekennzeichnet sind. Die Anzahl der Knoten im Graph ist dann proportional zur Anzahl der Mengen der Teilformeln von φ, $\mathcal{O}(2^{|\varphi|})$. Das Kreuzprodukt von Automaten $S \otimes S_{\neg\varphi}$ hat im ungünstigsten Fall die Komplexität $\mathcal{O}(2^{|\varphi|} \times |S|)$. Die Überprüfung der Leerheit eines Automaten ist linear, d.h. proportional zur Größe des Automaten. Daraus lässt sich herleiten, dass die Komplexität des LTL-Model Checking Verfahrens $\mathcal{O}(2^{|\varphi|} \times |S|^2)$ beträgt.

3.3.2 CTL-Model Checking

Das Problem, ob das Modell M die Formel φ erfüllt ($M \vDash \varphi$), geht das CTL-Model Checking folgendermaßen an. Die Erfüllbarkeit von φ wird überprüft, indem die Erfüllbarkeit aller Teilformeln von φ für jeden Zustand sukzessiv überprüft wird. Die Überprüfung fängt mit atomaren Aussagen an und endet mit der Formel φ. Als

nächstes wird eine Definition der Teilformeln einer CTL-Formel gegeben und der Algorithmus genauer beschrieben.

Definition 3.7. *(Teilformel einer CTL-Formel)*
Sei φ eine CTL-Formel über die atomaren Aussagen $p \in P$. Die Menge der Teilformeln von φ, $Sub(\varphi)$, wird folgendermaßen rekursiv gebaut:

$$\begin{aligned}
Sub(p) &= \{p\} \\
Sub(\neg\varphi) &= Sub(\varphi) \cup \{\neg\varphi\} \\
Sub(\varphi_1 \vee \varphi_2) &= Sub(\varphi_1) \cup Sub(\varphi_2) \cup \{\varphi_1 \vee \varphi_2\} \\
Sub(EX\varphi) &= Sub(\varphi) \cup \{EX\varphi\} \\
Sub(E[\varphi_1 U \varphi_2]) &= Sub(\varphi_1) \cup Sub(\varphi_2) \cup \{E[\varphi_1 U \varphi_2]\} \\
Sub(A[\varphi_1 U \varphi_2]) &= Sub(\varphi_1) \cup Sub(\varphi_2) \cup \{A[\varphi_1 U \varphi_2]\}
\end{aligned}$$

Die Markierung der Zustände lässt sich durch eine Funktion Sat beschreiben, die jeder CTL-Formel eine Menge der Zustände zuweist, die mit der Formel gekennzeichnet sind (siehe Tabelle 3.2). Die Zustandsmarkierung fängt mit der Markierung der Zustände an, in denen die atomaren Aussagen gültig sind. Dieser erste Schritt lässt sich mithilfe der Beschriftungsfunktion l beschreiben: Alle Zustände, in denen atomare Aussagen gültig sind, werden zunächst mit den Aussagen gekennzeichnet. Neben den atomaren Aussagen sind auch $\top$ und $\bot$ Formeln der Länge 1. $\top$ ist in allen Zuständen gültig und $\bot$ ($\neg\top$) ist in keinem Zustand gültig. Mit der Formel $\neg\varphi$ werden Zustände markiert, die noch nicht mit φ markiert sind. Im Fall von $\varphi_1 \vee \varphi_2$ wird überprüft, welche Zustände mit φ_1 oder φ_2 markiert sind. Diese Zustände werden auch mit $\varphi_1 \vee \varphi_2$ markiert. Ein Zustand von dem es eine Transition zum Zustand gibt, in dem φ gilt, kann mit $EX\varphi$ bezeichnet werden. Die Markierungen mit $E[\varphi_1 U \varphi_2]$ und $A[\varphi_1 U \varphi_2]$ sind etwas komplizierter und werden durch separate Algorithmen beschrieben (siehe Tabelle 3.2).

Mit $E[\varphi_1 U \varphi_2]$ werden zunächst Zustände markiert, die mit φ_2 bereits gekennzeichnet sind. In der nächsten Iteration werden mit $E[\varphi_1 U \varphi_2]$ Zustände markiert, die φ_1 erfüllen und von denen es Transitionen gibt, die zu mit φ_2 markierten Zuständen führen. Dieser Schritt wird wiederholt, solange es möglich ist, das heißt solange kein neuer Zustand die Bedingung erfüllt. Die Markierung mit $A[\varphi_1 U \varphi_2]$ erfolgt ähnlich. Man muss nur im Iterationsschritt mit $A[\varphi_1 U \varphi_2]$ Zustände markieren, die φ_1 erfüllen und von denen alle Transitionen zu mit φ_2 markierten Zuständen führen.

Wichtig bei den zwei Algorithmen ist eine Garantie, dass die Iterationsschritte terminieren. Ein Beweis dafür findet man in der Fixpunkt-Theorie. Der folgende Paragraph bietet einen Einblick in die Theorie.

Fixpunkt-Theorie

Für eine Abbildung $F : A \to A$ ist $a \in A$ ein Fixpunkt von F, wenn $F(a) = a$ gilt. Existenz eines Fixpunktes für bestimmte monotone Abbildungen wird durch einige Theoreme aus der Fixpunkt-Theorie gewährleistet. Beim CTL-Model Checking wird eine Abbildung $[\![]\!]$ folgendermaßen definiert: Für eine CTL-Formel φ wird mit $[\![\varphi]\!]$ die Menge der Zustände bezeichnet, in denen φ erfüllt ist:

$$[\![\varphi]\!] = \{q \in Q \mid \mathcal{M}, q \vDash \varphi\}.$$

Um die Menge der Zustände zu bestimmen, die mit $E[\varphi_1 \, U \, \varphi_2]$ markiert werden sollen, wird folgende Abbildung betrachtet:

$$F(Z) = [\![\varphi_2]\!] \cup ([\![\varphi_1]\!] \cup \{q \in Q \mid \exists q' \in Z.\ \delta(q, q')\}).$$

$[\![E[\varphi_1 \, U \, \varphi_2]]\!]$ ist ein Fixpunkt der Abbildung F. Ähnlich ist $[\![A[\varphi_1 \, U \, \varphi_2]]\!]$ der Fixpunkt der Abbildung:

$$G(Z) = [\![\varphi_2]\!] \cup ([\![\varphi_1]\!] \cup \{q \in Q \mid \forall q' \in Z.\ \delta(q, q')\}).$$

Komplexität

Beim CTL-Model Checking wird die Abbildung Sat für jede Teilformel der zu überprüfenden Formel ausgeführt. Die Anzahl der Teilformeln von φ ist proportional zur Formellänge $|\varphi|$. Andererseits wird die Formel für jeden Zustand überprüft (d.h. $|S|$-mal). Beim Sat_{AU} Algorithmus sollen für jeden Zustand alle Transitionen, die aus dem Zustand gehen, durchgelaufen werden. Diese Zahl ist im ungünstigsten Fall proportional zu $|S|^2$. Das heißt, dass die Komplexität vom CTL-Model Checking $\mathcal{O}(|\varphi| \times |S|^3)$ beträgt. Diese Komplexität kann verbessert werden, wenn CTL durch EX, $E[\varphi_1 U \varphi_2]$ und EG definiert wird. In diesem Fall lässt sich die Komplexität auf $\mathcal{O}(|\varphi| \times |S|^2)$ reduzieren ([CES86]).

function $Sat(\varphi : \ CTL - Formel) : \ Zustandsmenge$
begin
  switch(φ)
    $\top$: **return** Q
    p: **return** $\{q \mid \varphi \in l(q)\}$
    $\neg\psi$: **return** $Q \setminus Sat(\psi)$
    $\psi_1 \vee \psi_2$: **return** $Sat(\psi_1) \cup Sat(\psi_2)$
    $EX\psi$: **return** $\{q \in Q \mid \exists q' \in S.\ \delta(q, q') \ \wedge \ q' \in Sat(\psi)\}$
    $E[\psi_1 \ U \ \psi_2]$: **return** $Sat_{EU}(\psi_1, \psi_2)$
    $A[\psi_1 \ U \ \psi_2]$: **return** $Sat_{AU}(\psi_1, \psi_2)$
  end switch
end

---

function $Sat_{EU}(\varphi_1, \varphi_2 : \ CTL - Formel) : \ Zustandsmenge$
var
  $Z, Z' : \ Zustandsmenge$
begin
  $Z := Sat(\varphi_2)$, $Z' := \varnothing$
  while
    $Z' := Z$
    $Z := Z \cup (\{q \in Q \mid \exists q' \in Z.\ \delta(q, q')\} \cap Sat(\varphi_1))$
  end while
  return Z
end

---

function $Sat_{AU}(\varphi_1, \varphi_2 : \ CTL - Formel) : \ Zustandsmenge$
var
  $Z, Z' : \ Zustandsmenge$
begin
  $Z := Sat(\varphi_2)$, $Z' := \varnothing$
  while
    $Z' := Z$
    $Z := Z \cup (\{q \in Q \mid \forall q' \in Z.\ \delta(q, q')\} \cap Sat(\varphi_1))$
  end while
  return Z
end

Tabelle 3.2: a) Basisalgorithmus fürs CTL-Model Checking, b) Markierung mit der Formel $E[\varphi_1 \, U \, \varphi_2]$, c) Markierung mit der Formel $A[\varphi_1 \, U \, \varphi_2]$

3.4 Verifikation von SPS (Stand der Technik)

In den letzten Jahrzehnten wurden erhebliche Fortschritte in der formalen Verifikation von SPS erreicht. [Mad00, FL98, Huu03] sind einige der Studien, die einen Überblick über die existierenden Verfahren bieten. Einige der Verfahren werden unter anderem in den folgenden Abschnitten beschrieben. Eine tabellarische Darstellung der untersuchten Veröffentlichungen ist in Tabelle 3.3 angegeben. Die Tabelle gibt eine Klassifizierung bezüglich der folgenden drei Kriterien: 1) SPS-Sprache, 2) Formalisierungsmodell und 3) Verifikationsmethode.

3.4.1 AWL-Verifikation

Wenn Echtzeiteigenschaften bei SPS-Anwendungen berücksichtigt werden sollen, sind Timed Automata für die Modellierung geeignet. In [MW99] wird beschrieben, wie man AWL-Programme mit Timed Automata modellieren kann. Aufrufe von Funktionen und Funktionsbausteinen werden dabei nicht berücksichtigt. Als Variablen dürfen nur Booleans verwendet werden. Basierend auf die Arbeit wurde ein Tool entwickelt, das AWL-Programme in Timed Automata übersetzt ([Wil99]). Für die Verifikation wird der Model Checker UPPAAL benutzt.

Eine formale Semantik von AWL ist in [Huu03] veröffentlicht. In der Arbeit wird vorgeschlagen, eine Programmanalyse in zwei Schritten durchzuführen. Zuerst wird eine Approximation des Programmverhaltens durch abstrakte Interpretation gemacht ([Cou78, CC79]), um beispielsweise Bereiche der Programmvariablen während Programmausführung zu bestimmen. Danach wird eine Datenflussanalyse durchgeführt ([Hec77]), um beispielsweise den Informationsfluss zu untersuchen oder nicht erreichbaren Code zu entdecken. Das begleitende Tool für die Unterstützung der Programmanalyse (Homer) wird in [Huu04] beschrieben.

Trotz des Ziels, eine Analyse von F-FUP-Programmen zu schaffen, kann man auch [Fig06] an dieser Stelle erwähnen. In der Arbeit wird beschrieben, wie man AWL-Code nach VHDL transformieren kann. VHDL selbst bietet weitere Verifikationsmöglichkeiten. Der Schwerpunkt der Arbeit liegt allerdings in der Simulation und Automatisierung der Testumgebung.

In [HM98] werden AWL Programme mit Petrinetzen modelliert und mit dem Model Checker SMV verifiziert. Dabei werden nur Datenstrukturen benutzt, die mit 8 Bits codiert werden können.

Ein weiteres Verfahren für AWL-Verifikation mit dem Model Checker SMV wird in [CCLP00] beschrieben. Dafür werden Programme durch Transitionssystemen modelliert. Zeit und Timer werden in der Arbeit nicht berücksichtigt. Das Verfahren wurde als Basis für [PPKE07] genommen und dort weiterentwickelt. Die Automatisierung des Verfahrens und die begleitenden Tools sind in [PPK07] beschrieben.

Der SMV Model Checker wird für die AWL-Verifikation auch in [LYF05] benutzt. Dabei wird ein AWL-Programm zuerst anhand eines Mealy Automaten formalisiert ([YF04]). Den Mealy Automat kann man zu einem Moore Automat transformieren, den man als ein SMV-Modell darstellen kann. Bei der Verifikation durch SMV werden auf BDDs und auf SAT basierte Verifikationstechniken verglichen. Als Ergebnis stellt man fest, dass soweit keine Zustandsexplosion stattfindet, wird BDD-Model Checking bevorzugt.

3.4.2 ST-Verifikation

Wenn man formale Verifikation beim Steuerungsentwurf anwenden möchte, ist es möglich, die SPS-Spezifikation in einer Form vom Duration Calculus anzugeben und diese als einen PLC-Automaten darzustellen ([Die97]). Diese Art von Automaten ist laut [Die98] nach ST automatisch übersetzbar. In dem Projekt wurde die Anwendbarkeit der zwei Model Checker KRONOS und UPPAAL auf PLC-Automaten getestet.

Ein auf HOL (Higher-Order Logic) basierndes Verfahren für ST-Verifikation wird in [VK97, VK99, Vö00] veröffentlicht. AS-Programme werden durch HOL-Automaten dargestellt und mit dem generischen Theorem Prover Isabelle verifiziert. In der Arbeit wird die Verifikation von hybriden Systemen betrachtet. Neben der AS werden auch ST und FBS Programme berücksichtigt.

Ein Verfahren für die ST-Verifikation mit dem Model Checker NuSMV wird in [GSF06, GSF08] beschrieben. Bei der Konstruktion eines NuSMV Modells wird nicht jede Programmlinie einzeln betrachtet. Es wird zuerst eine Analyse der Variablenabhängigkeiten durchgeführt. Dabei wird betrachtet, inwiefern Ausgangsvariable von Eingangsvariablen abhängig sind. Abhängig von den Ergebnissen dieser Analyse wird ein NuSMV-Modell erstellt, das deutlich kleiner ist als das Ergebnissmodell einer direkten Übersetzung von ST nach NuSMV. Das Verfahren lässt sich erweitern und auch auf weitere SPS-Sprachen anwenden. Allerdings gibt es zwei große Einschränkungen bei der Art der Verifikation: 1) Es können nur boolesche Variablen

benutzt werden; und 2) Sprünge sind im Programm nicht erlaubt.

3.4.3 FBS Verifikation

In Unterkapitel 3.4.1 wurde bereits das in [Fig06] veröffentlichte Verfahren erwähnt. In der Arbeit wird eine Analyse von einer Untermenge der AWL-Programme vorgeschlagen. Der Schwerpunkt der Arbeit liegt in der Analyse von F-FUP-Programmen. Es wird dabei die AWL Interpretation von F-FUP-Programmen benutzt. Diese wird nach VHDL übersetzt. In der Arbeit wird nicht weiter die Verifikation, sondern eher die Simulation und Automatisierung der Testumgebung behandelt.

In den letzten fünf Jahren wurden große Fortschritte bei der FBS-Verifikation in Korea im Bereich von Kernkraftwerken gemacht. In dem Projekt wird FBS sowohl für die Software Designspezifikation als auch für die Implementierung benutzt ([SKS04, Son03]). Bei der Spezifikation werden Entscheidungstabellen für die Anforderungen und Design von Software erstellt. Anschließend werden diese zwei Tabellen verglichen. Positiv dabei ist, dass man die FBS-Designspezifikation nutzen kann, um Zeit und Kosten bei der Implementierung zu sparen. Negativ anzumerken ist, dass, wenn die Spezifikation kompliziert ist, man sie nicht in einer Entscheidungstabelle behandeln kann. Andererseits wird Software durch den Model Checker SMV verifiziert. Ein FBS-Programm wird dabei manuell durch Übersetzungsregeln nach SMV transformiert. Die neusten Ergebnisse aus dem Projekt sind in [Jeo06], [KSJ+07] und [KJJ+08] veröffentlicht. Die Transformation von FBS nach SMV läuft mittlerweile automatisch. Ein FBS-Programm wird zuerst nach Verilog und dann nach SMV transformiert. Im Unterschied zum ersten Beispiel in diesem Kapitel, wo ein VHDL-Modell auf Basis der AWL-Interpretation eines FBS-Programms konstruiert wurde, wird in diesem Projekt das Verilog-Modell auf FBS-Basis dargestellt. Wie man ein Verilog-Modell mit SMV verifiziert, ist bereits bekannt. Interessanter ist es, wie man von einem FBS-Programm ein Verilog Model konstruiert. Für diesen Zweck wird ein Tool in der Entwicklung, pSET, benutzt. Leider ist das Format (*.lda), in dem pSET ein FBS-Programm speichert, nicht bekannt.

3.4.4 KOP-Verifikation

Ein Verfahren für die KOP-Verifikation wird in [RS00] vorgestellt. Eine Untermenge der KOP-Konstrukte wird betrachtet. Für die Modellierung werden Zustandsauto-

maten benutzt. Verifikation wird durch den Cadence SMV Model Checker durchgeführt. Es werden logische Aspekte der Zeit betrachtet, jedoch keine Echtzeit. Damit wird die Komplexität der Timed Automata vermieden und trotzdem ist es möglich, Eigenschaften zu überprüfen, die sich auf Timer beziehen.

In [RD02] wird beschrieben, wie man KOP-Programme durch Theorem Proving verifizieren kann. Für die Modellierung wird eine Algebra definiert, durch die eine algebraische Semantik den KOP-Programmen zugewiesen wird. Einer der Autoren untersucht in [ZRK03], wie sich Model Checking auf KOP-Programmen anwenden lässt. Für die Modellierung werden Timed Automata benutzt. Die Automaten werden mit dem Model Checker UPPAAL verifiziert. Das Verifikationsverfahren wurde automatisiert.

3.4.5 AS-Verifikation

In [HIZ98] werden hybride Automaten benutzt, um einerseits ein AS-Programm und andererseits den kontrollierten Prozeß darzustellen. Model Checking des kombinierten Modells wird mit dem Tool HyTech durchgeführt.

Das in [VK99, Vö00] veröffentlichte Verfahren für SPS-Verifikation wird bereits in Unterkapitel 3.4.2 erwähnt. Mit dem Verfahren lassen sich auch AS-Programme verifizieren, in dem sie erst nach ST und danach nach HOL transformiert werden.

Ein weiteres Verfahren für die AS-Verifikation schlägt vor, Transitionssysteme für die AS-Modellierung zu nutzen ([HLB03]). Die Systeme werden durch den Cadence SMV Model Checker verifiziert. Die Arbeit weist darauf hin, dass die IEC Regel für die AS-Programmierung zur Formulierung von nichtsicheren Programmen führen können. Dabei wird eine Definition von *sicheren AS-Programmen* in der Arbeit gegeben.

3.4.6 Andere Verfahren

Neben den Verfahren für die Verifikation der Programme, welche in einer konkreten SPS-Sprache geschrieben sind, ist es interessant zu sehen, wie sich Verifikation beim Entwurf von Steuerungen anwenden lässt. In [FL98] werden für die Modellierung besondere Petrinetze benutzt: 1) steuerungstechnisch interpretierte Petrinetzen (SIPN) werden für die Modellierung des steuernden Systems, also des Steuerungsalgorithmus, benutzt; 2) prozeßbezogene interpretierte Petrinetze (PIPN) werden

benutzt, um den zu steuernden Prozeß (einschließlich Aktorik und Sensorik) zu modellieren ([FL99]). Verifikation wird dann durch Erreichbarkeitsanalyse vom kombinierten Modell durchgeführt. In [WL00] wird beschrieben, wie sich SIPNs durch Model Checking verifizieren lassen. Ein Tool für die automatische Erzeugung einer Kripke-Struktur aus SIPNs wurde entwickelt ([Lit05]). Ein anderes Verfahren für die Verifikation von SIPNs schlägt vor, aus einem SIPN-Modell erst AWL-Code zu generieren ([MF01, MM00]). Der AWL-Code wird in ein anderes Petrinetz übersetzt, das mit dem Petrinetz der SPS und der Umgebung kombiniert und durch Model Checking verifiziert wird. Die Transformation von AWL nach SMV wird bereits in Abschnitt 3.4.1 erwähnt.

Beim EU VHS Projekt (Verification of Hybrid Systems) geht es um den Entwurf und die Verifikation einer experimentellen chemischen Anlage. In [MBWB01] wird gezeigt, wie ein Kontrollprogramm schrittweise durch die Verwendung eines echtzeitlogischen Formalismus entworfen werden kann. Als Verifikationsmethode werden sowohl Theorem Proving (PVS) als auch Model Checking verwendet (SPIN). Das gleiche Fallbeispiel wird auch mit UPPAAL verifiziert ([BMF02]). Die Model Checking Ergebnisse mit SPIN und UPPAAL werden verglichen.

Sprache	Referenz	Model	Methode
AWL	[MW99], [Wil99]	Timed Automata	Model Checking
	[HM98]	Petrinetz	Model Checking
	[Huu03], [Huu04]	Transitionssystem	Programmanalyse
	[CCLP00], [PPKE07], [PPK07]	Transitionssytem	Model Checking
	[LYF05], [YF04]	Mealy und Moore Automata	Model Checking
	[Fig06]	VHDL	Simulation
ST	[Die97]	PLC-Automata	Model Checking
	[VK97], [VK99], [Vö00]	HOL	Theorem Proving
	[GSF06], [GSF08]	Transitionssystem	Model Checking
FBS	[Fig06]	VHDL	Simulation
	[Jeo06], [KSJ+07], [KJJ+08]	Transitionssystem	Model Checking
KOP	[RS00]	Zustandsautomata	Model Checking
	[RD02]	Boolesche Algebra	Theorem Proving
	[ZRK03]	Timed Automata	Model Checking
AS	[HIZ98]	Hybride Automaten	Model Checking
	[KP96]	Timed Condition/Events	Model Checking
	[HLB03]	Transitionssystem	Model Checking
	[VK99], [Vö00]	HOL	Theorem Proving
Diverses	[FL98]	Petrinetz	Erreichbarkeitsanalyse
	[WL00]	Kripke-Struktur	Model Checking
	[MBWB01], [BMF02]	Echtzeitlogik	Model Checking, Theorem Proving

Tabelle 3.3: Übersicht über die untersuchten Verfahren für SPS-Verifikation

4 Semantik

Es gibt zwei wichtige Gründe für Formalisierung von SPS (siehe Abbildung 4.1). Der Erste ist der Bedarf an Anwendung einer der Methoden, die Qualität der SPS-Software nachweisen. Einige Beispiele dafür sind Verifikation, Validierung, Simulation oder Analyse. Die Methoden lassen sich leichter anwenden, wenn ein formales Model der SPS schon vorhanden ist. Der zweite Grund für die SPS-Formalisierung liegt in ständiger Weiterentwicklung von SPS und der Umgebung, in der sie angewendet werden. Dabei sollten beispielsweise schon existierende Programme neu angepasst werden oder auf eine Hardwareplattform umgezogen werden. Sehr oft wird diese Aufgabe durch eine komplett neue Programmierung der Steuerung gelöst.

SPS-Programmiersprachen sind zur leichteren SPS-Programmierung in der Norm IEC 61131-3 ([Int03]) standardisiert. In der Norm sind die Sprachen zwar beschrieben werden aber nicht formal definiert. Eine formale Definition der Syntax und Semantik von AWL und AS wurde von Huuck in [Huu03] vorgestellt, während ST von Völker in [Vö98] formal beschrieben wurde. In dieser Arbeit wird die grafische Programmiersprache FBS betrachtet. Die Norm IEC 61131-3 stellt keine vollständige

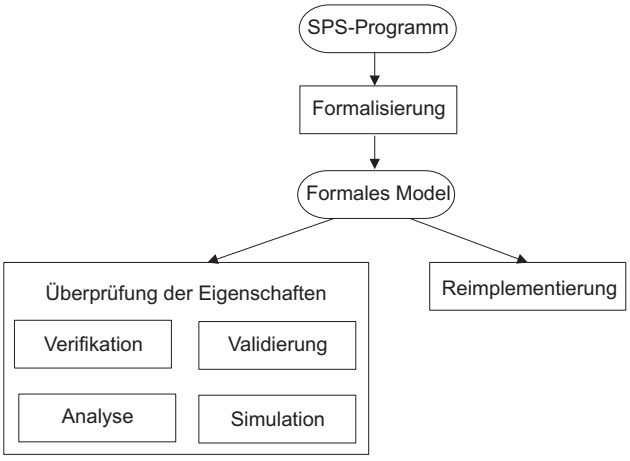


Abbildung 4.1: SPS-Formalisierung

Liste der FBs-Anweisungen zur Verfügung, sodass sich eine detaillierte Beschreibung der Sprache erst bei der Programmierung konkreter SPS finden lässt. In dieser Arbeit wurde die Sprache FUP (Funktionsplan) für die SIMATIC-Steuerung betrachtet (siehe [Sie04b]).

In diesem Kapitel wird die formale Beschreibung von FUP vorgeschlagen. Zuerst werden einige Techniken für die Semantikbeschreibung vorgestellt und danach werden Syntax und Semantik eines FUP-Programms definiert.

4.1 Ansätze

Unter Semantik versteht man ursprünglich ein Teilgebiet der Linguistik in dem man sich mit der „Lehre von der Bedeutung sprachlicher Zeichen“ ([Feh89]) befasst. In der Informatik wird unter Semantik meistens die Bedeutung eines Programms verstanden. Allgemeiner betrachtet bezieht sich Semantik auf die Bedeutung syntaktischer Konstrukte in der Informatik. Abhängig vom Kontext versteht man darunter im täglichen Sprachgebrauch Folgendes ([Kin05]):

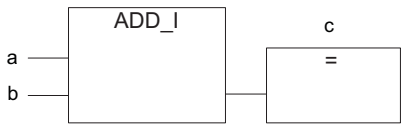
Semantik eines Programms: Bedeutung eines konkreten Programms.

Semantik einer Programmiersprache: Eine Zuordnung der Bedeutung jedem syntaktisch korrekten Programm einer konkreten Programmiersprache.

Semantik von Programmiersprachen: Techniken zur Untersuchung der Bedeutung verschiedenartiger Programmiersprachen.

Es gibt mehrere Techniken, mit denen sich einem Programm eine Bedeutung zuordnen lässt. Operationale, mathematische und axiomatische Semantik sind einige von ihnen, deren Hauptmerkmale in diesem Kapitel erläutert werden. Die Eigenschaften der drei Ansätze werden am folgenden kleinen FUP-Beispiel erklärt.

Beispiel 4.1. *($c = a + b$)*
Ein Beispiel in dem zwei Variablen addiert werden und das Ergebnis einer dritten Variable zugewiesen wird, lässt sich in FUP folgendermaßen darstellen:



In weiterer Anwendung des Beispiels kann folgende textuelle Interpretation be-

nutzt werden:

$$p \quad \equiv \quad \_L := (a + b); \quad c := \_L;$$

wobei $\_L$ *eine temporäre Variable ist, die das Zwischenergebnis speichert.*

4.1.1 Operationale Semantik

Unter operationaler Semantik versteht man eine Zuordnung, die einem konkreten Programm eine Bedeutung zuweist, wobei die Bedeutung des Programms schrittweise definiert wird. Um das zu erläutern, wird vorerst der Begriff des Zustands näher erläutert.

Unter einem Zustand versteht man eine Belegung der Programmvariablen. Dabei ist es erlaubt, dass eine Variable einen undefinierten Wert ϵ hat. Für das Programm p aus Beispiel 4.1 kann zum Beispiel ein Zustand definiert werden, in dem die Variablen folgende Werte haben: $a = 1$, $b = 2$, $\_L = \epsilon$ und $c = \epsilon$. Dieser Zustand wird notiert als $[a/1,\, b/2,\, \_L/\epsilon,\, c/\epsilon]$. Die Abarbeitung von p lässt sich dann als Zustandsfolge wie folgt repräsentieren:

$$\langle p,\, [a/1,\, b/2,\, \_L/\epsilon,\, c/\epsilon]\rangle \rightarrow \langle \_L := (a + b);\, p,\, [a/1,\, b/2,\, \_L/\epsilon,\, c/\epsilon]\rangle \rightarrow$$
$$\langle p,\, [a/1,\, b/2,\, \_L/3,\, c/\epsilon]\rangle \rightarrow \langle c := \_L;\, p,\, [a/1,\, b/2,\, \_L/3,\, c/3]\rangle \rightarrow$$
$$\langle p,\, [a/1,\, b/2,\, \_L/3,\, c/3]$$

4.1.2 Mathematische (denotationale) Semantik

Eine Eigenschaft der operationalen Semantik ist es, dass die Beschreibung des Verhaltens eines Programms von einem konkreten Startzustand ausgeht. Im Gegensatz zur operationalen Semantik, wird bei der mathematischen Semantik eine allgemeine Beschreibung angeboten, die ohne Bezug auf konkrete Zustände auskommt. Die mathematische Semantik ist eine partielle Abbildung, $[\![p]\!] \colon \Sigma \rightharpoonup \Sigma$, die jedem Startzustand einen entsprechenden Endzustand zuweist, in dem das Programm terminiert. Falls das Programm nicht terminiert, ist das Ergebnis der Funktion nicht definiert.

Das oben aufgeführte Beispielprogramm terminiert immer. Da das Programm vier Variablen (a, b, $\_L$, c) beinhaltet, kann man den Zustand als Quadrupel ganzer Zahlen betrachten, die die Werte der Variablen darstellen. Demzufolge lässt sich die

mathematische Semantik folgendermaßen definieren:

$$[\![p]\!] \colon Z \times Z \times Z \times Z \to Z \times Z \times Z \times Z$$
$$(a, b, \_L, c) \mapsto (a, b, a + b, a + b)$$

4.1.3 Axiomatische Semantik

Drittes Beispiel für die Beschreibung des Programmverhaltens ist die axiomatische Semantik. Mit der Semantik werden bestimmte Vor- und Nachbedingungen als Programmeigenschaften formuliert. Diese Eigenschaften werden auch *Zusicherungen* genannt. So kann man im Beispiel 4.1 Folgendes behaupten: Wenn vor der Programmausführung die Variable a den Wert m hat und die Variable b den Wert n, dann wird die Variable c am Programmende den Wert $m+n$ haben. Wenn vorausgesetzt wird, dass Variablen a, b und c vom Typ *integer* sind, lassen sich folgende Zusicherungen definieren:

$$\{a = m \wedge b = n\} \quad \_L := (a + b); \quad c := \_L; \quad \{c = m + n\}$$

Damit wird mittels axiomatischer Semantik Bedeutung eines Programms mit Hilfe seiner Eigenschaften implizit beschrieben.

4.2 Formalisierung der Sprache Funktionsplan

Es wird im Allgemeinen angenommen, dass die für die SPS-Programmierung grundlegende Sprache die maschinenorientierte Sprache AWL ist. Alle anderen SPS-Sprachen lassen sich in AWL abbilden. Es ist dabei erwähnenswert, dass der Parallelismus bei AS die einzige Eigenschaft einer Sprache ist, die sich in AWL nicht darstellen lässt ([Huu03]).

In diesem Abschnitt wird zuerst die Sprache FUP näher betrachtet. Anschließend wird die Formalisierung der Syntax und der Semantik eingeführt.

4.2.1 FUP

Im Allgemeinen ist FUP eine eingeschränkte graphische Repräsentation von AWL. Als Beispiel wird in Abbildung 4.2 der Vergleich von zwei Integerwerten in beiden

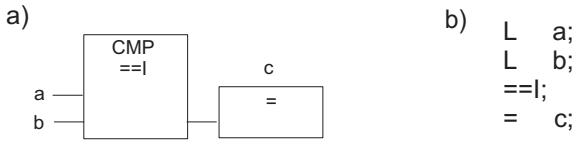


Abbildung 4.2: Vergleich von zwei Integer: a) FUP, b) AWL

Sprachen dargestellt. Elemente der FUP-Sprache sind a) Variablen und Konstanten, b) Funktionen und Funktionsbausteine und c) Verbindungen zwischen diesen Elementen.

FUP-Anweisungen

Es gibt verschiedene Arten von FUP-Anweisungen, die im Folgenden aufgelistet sind. Für jeden Anweisungstyp ist in Tabelle 4.1 jeweils ein Beispiel angegeben.

- Die Bitverknüpfungsoperationen verknüpfen ihre Eingänge entsprechend der Booleschen Logik und liefern als Ergebnis 1 oder 0, das sogenannte Verknüpfungsbit *VKE*.

- Vergleichsoperationen haben zwei Eingänge (Ganz- oder Gleitpunktzahlen), die entsprechend der folgenden Vergleichsarten verglichen werden: gleich, ungleich, größer, größer oder gleich, kleiner und kleiner oder gleich.

- Sprünge in Codebausteinen werden durch Sprungoperationen realisiert. Ein Sprung kann absolut oder unter einer Bedingung durchgeführt werden.

- Mit den Festpunkt-Funktionen können zwei Ganzzahlen (16 oder 32 Bit) addiert, subtrahiert, multipliziert oder dividiert werden. Zusätzlich lässt sich der Divisionsrest von zwei 32 Bit Ganzzahlen bestimmen.

- Verschieben wird mit der $MOVE$ Operation dargestellt, mit der der am Eingang IN angegebene Wert, in den am Ausgang OUT angegebenen Operanden kopiert wird.

- Zu den Programmsteuerungsoperationen gehören verschiedene Operationen, mit denen Aufrufe von Funktionen und Funktionsbausteinen realisiert werden.

- Umwandlungsoperationen ermöglichen die Umwandlung von binär-codierten Dezimalzahlen und Ganzzahlen in andere Zahlenarten. Zu den Operationen

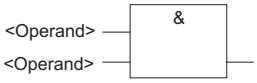
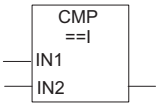
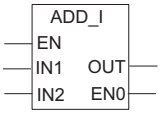
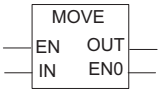
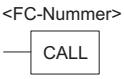
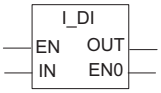
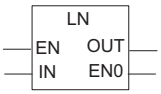
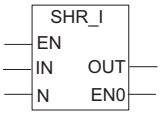
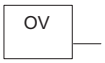
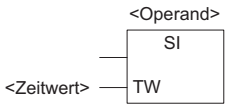
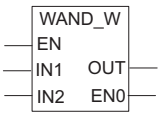
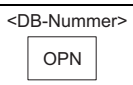
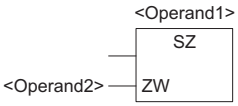
Typ	Beispiel	
Bitverknüpfung		UND-Verknüpfung
Vergleicher		Ganzzahlen vergleichen
Sprünge		Springe im Baustein absolut
Festpunkt-Funktionen		Ganzzahlen addieren
Verschieben		Wert übertragen
Programmsteuerung		Funktion aufrufen
Umwandler		Ganzzahlen umwandeln (16 in 32 Bit)
Gleitpunktfunktionen		Bilden des natürlichen Logarithmus'
Schieben/Rotieren		Ganzzahl rechts schieben
Statusbits		Störungsbit Überlauf
Zeiten		Zeit als Impuls starten
Wortverknüpfung		16 Bit UND verknüpfen
DB-Aufruf		Datenbaustein öffnen
Zähler		Zähleranfangswert setzen

Tabelle 4.1: Beispiele von FUP-Funktionen und Funktionsbausteine

gehört auch die Erzeugung des Komplements einer Ganzzahl oder die Umwandlung einer Gleitpunktzahl in eine Ganzzahl zum Beispiel durch Aufrunden.

- Zu den Gleitpunktfunktionen gehören verschieden arithmetische und trigonometrische Funktionen mit Gleitpunktzahlen.
- Der Inhalt eines Eingangs lässt sich bitweise nach links oder rechts verschieben oder rotieren. Das wird mittels Schiebe- oder Rotieroperation gemacht.
- Statusbitoperationen sind Bitverknüpfungsoperationen, mit denen Bits des Statusworts angesprochen werden.
- Verschiedene Zeitoperationen stehen zum Einstellen und zur Auswahl der richtigen Zeit zur Verfügung.
- Wortverknüpfungsoperationen verknüpfen zwei Eingänge entsprechend der Booleschen Logik.
- Mit der OPN Operation lässt sich ein Datenbaustein als globaler oder Instanz-Datenbaustein öffnen.
- Zähler haben einen eigenen reservierten Speicherbereich in der CPU. Zu den Zähloperationen gehören Operation wie Zähler parametrieren, Zähleranfangswert setzen, vorwärts- oder rückwertszählen.

EN-/ENO-Mechanismus

Bei manchen Elementen in Tabelle 4.1 erscheinen bestimmte EN und ENO Bits am Eingang bzw. Ausgang. Das sind Bits für die Freigabe (EN) und für den Freigabeausgang (ENO) der FUP-Boxen. Wenn EN und ENO beschaltet sind und die Information über Fehler in der Box in $Fehler$ aufbewahrt wird, dann gilt:

$$ENO = EN \wedge \neg Fehler$$

Wenn kein Fehler auftritt ($Fehler = 0$), ist somit $ENO = EN$. Der EN-/ENO-Mechanismus wird verwendet für:

- arithmetische Operationen,

- Übertragungs- und Umwandlungsoperationen,
- Schiebe- und Rotieroperationen,
- Bausteinaufrufe.

Der Mechanismus wird nicht verwendet für:

- Vergleicher,
- Zähler,
- Timer.

4.2.2 FUP-Syntax

FUP unterstützt verschiedene Kategorien von Parametern. Die Parameter sind eingeteilt in: Eingangs-, Ausgangs-, Ein- und Ausgangs-, statischen oder temporären Variablen. Werte dieser Variablen können verschiedene Datentypen haben (bspw. Boolean oder Integer). Die Variablenkategorien und Datentypen werden bei der Formalisierung der FUP-Syntax in dieser Arbeit abstrahiert, indem gesagt wird, dass FUP-Variablen verschiedenen Datentypen angehören.

Sei V die Menge der Programmvariablen. Die Vereinigung der Wertebereiche aller Variablen aus V wird mit D bezeichnet. Wie bereits erwähnt, gehören zu den Elementen eines FUP-Programms auch Verbindungen zwischen Funktionen und Funktionsbausteinen. Um diese Verbindungen zu beschreiben, wird die Klasse von Leitungsvariablen $W = \{\_Li \mid i \in \mathbb{N}\}$ eingeführt.

Sei Id die Menge der Bezeichnungen (Namen) der FUP-Bausteine (siehe das Handbuch „SIMATIC - Funktionsplan (FUP) für S7-300/400“ [Sie04b]).

Definition 4.1. *(FUP-Baustein)*
Ein Baustein ist ein Tupel (id, In, Out, Loc)*, dessen Komponenten wie folgt definiert sind:*

- $id \in Id$ *ist die Bezeichnung (Name) eines Bausteins,*
- In *ist ein Tupel der Eingänge des Bausteins,* $In = (in_1, \ldots, in_p)$*,* $in_i \in D \cup V \cup W$*,* $1 \leq i \leq p$*,* $p \in \mathbb{N}$*,*

- *Out ist ein Tupel der Ausgänge des Bausteins,* $Out = (out_1, \ldots, out_q)$, $out_i \in V \cup W$, $1 \leq i \leq q$, $q \in \mathbb{N}$, *und*

- $Loc \subset V$ *ist die Menge der lokalen Variablen, die weder Eingangs-, noch Ausgangsvariable im Baustein sind.*

Definition 4.2. *(Netzwerk)*
Ein Netzwerk ist eine Menge von FUP-Bausteinen $N = \{e_i \mid 1 \leq i \leq n\}$, $n \in \mathbb{N}$. *Verbindungen zwischen Bausteinen werden durch Leitungsvariablen realisiert, die einen Datenfluss zwischen Elementen beschreiben. Ein Netzwerk hat folgende Eigenschaften:*

- *Es gibt keine freien Leitungen im Netzwerk. Zudem ist die durch die Leitungen realisierte Datenfluss-Relation zwischen zwei Elementen im Netzwerk eine 1:n Relation. Es heißt:*

$$\begin{aligned}(\forall e \in N)\,(\forall j)\quad (out_j^e \in W \rightarrow\ & (\exists e' \in N)\,(e' \neq e)\,(\exists k)\, out_j^e = in_k^{e'}) \ \wedge \\ (in_j^e \in W \rightarrow\ & (\exists e' \in N)\,(e' \neq e)\,(\exists k)\, in_j^e = out_k^{e'} \ \wedge \\ & \neg((\exists e'' \in N)\,(e'' \neq e')\,(e'' \neq e)\,(\exists k)\, in_j^e = out_k^{e''}))\end{aligned}$$

- *Zwei beliebige Elemente im Netzwerk sind stets verbunden:*

$$(\forall e, e' \in N)\,(\exists k \leq n)\,(\exists e_1, \ldots, e_k \in E)\, e = e_1 \wedge e' = e_k \wedge$$
$$(\forall i < k)(\exists \_\, L_i \in W)(\exists j)(\exists l) \_\, L_i = out_j^{e_i} = in_l^{e_{i+1}} \veebar \_\, L_i = in_j^{e_i} = out_l^{e_{i+1}}$$

  Es wird geschrieben $e \sim e'$. *Wenn für* e *und* e' *in der vorherigen Bedingung immer* $\_\, L_i = out_j^{e_i} = in_l^{e_{i+1}}$ *gilt, sagt man, dass* e' *ein Nachfolger von* e *im Netzwerk ist* ($e \prec e'$).

- *es gibt keine rekursiven Verbindungen im Netzwerk. Das heißt, die transitive Hülle von* $\prec$ ($\prec^+$) *ist eine asymmetrische Relation.*

Die wichtigste Eigenschaft für den Datenfluss zwischen zwei Elementen e und e' ist meist, ob überhaupt eine Verbindung zwischen e und e' besteht. Dabei ist es irrelevant, zwischen welchen Variablen genau die Daten fließen. Demzufolge wird $out^e = in^{e'}$ vorausgesetzt.

Das in Beispiel 4.1 gegebene Netzwerk besteht aus zwei Elementen, e_1 und e_2, die folgendermaßen definiert sind: $e_1 = (CMP == I, (a, b), (\_L1), \varnothing)$ und $e_2 = (=, (\_L1), \varnothing, \{c\})$.

Definition 4.3. *FUP-Programm*
Ein Programm ist ein Tupel $\mathcal{P} = (\mathcal{N}, <)$, wobei

- *$\mathcal{N}$ eine endliche Menge von Netzwerken ist, und*
- *$< \subset \mathcal{N} \times \mathcal{N}$ eine irreflexive totale Ordnung ist, die die Reihenfolge von Netzwerken im Programm definiert.*

4.2.3 FUP-Semantik

Wie bereits in Abschnitt 4.1 erklärt, versteht man unter dem Begriff Semantik eine Abbildung eines syntaktisch korrekten Programms auf dessen Bedeutung. Von den bereits erwähnten Semantiken wird für die Definition von FUP-Semantik die operationale Semantik verwendet und das Verhalten eines FUP-Programms schrittweise definiert. Es wird aber davor eine Definition des bereits eingeführten Begriffs Zustand gegeben und die Änderungen des Programmzustands näher betrachtet.

Zustand

Definition 4.4. *(Zustand)*
Der Zustand eines Programms ist eine partielle Abbildung, $\sigma \colon V \rightharpoonup D$, die jeder Variable aus V höchstens einen Wert aus ihrem Wertebereich zuweist. Ein undefinierter Wert einer Variable wird durch ϵ bezeichnet. Die Menge aller Zustände wird durch Σ bezeichnet.

Abhängig vom aktuellen Zustand σ und vom auszuführenden Element e erhalten die Programmvariablen durch die Ausführung des Elements neue Werte, die in einem neuen Zustand σ' zusammengefasst sind (siehe Tabelle 4.2). Formal wird dies notiert als: $\sigma' = e(\sigma)$. Die entsprechenden operationalen Regeln werden in Tabelle 4.2 gezeigt. Beispielsweise wird als erste Bitverknüpfungsoperation in der Tabelle die UND-Verknüpfung folgendermaßen beschrieben: wenn das Element e zwei Eingänge in_1 und in_2 und einen Ausgang *out* hat $(e = (\&, (in_1, in2), (out), \varnothing))$ und wenn die Ausführung des Elements die Auswirkung hat, dass der Ausgang *out* der logischen Verknüpfung $\sigma(in_1) \wedge \sigma(in_2)$ entspricht, dann gilt $\sigma' = e(\sigma)$.

Bitverknüpfung

$$\frac{e=(\&,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=\sigma(in_1)\wedge\sigma(in_2)}{\sigma'=e(\sigma)}\,\&$$

$$\frac{e=(>=1,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=\sigma(in_1)\vee\sigma(in_2)}{\sigma'=e(\sigma)}\,>=1$$

$$\frac{e=(XOR,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=\sigma(in_1)\veebar\sigma(in_2)}{\sigma'=e(\sigma)}\,XOR$$

$$\frac{e=(=,(in),\varnothing,\{loc\}) \quad \sigma'(loc)=\sigma(in)}{\sigma'=e(\sigma)}\,=$$

$$\frac{e=(\#,(in),(out),\{loc\}) \quad \sigma'(loc)=\sigma(in) \quad \sigma'(out)=\sigma(in)}{\sigma'=e(\sigma)}\,\#$$

$$\frac{e=(R,(in),\varnothing,\{loc\}) \quad in\rightarrow\neg\sigma'(loc)}{\sigma'=e(\sigma)}\,R$$

$$\frac{e=(S,(in),\varnothing,\{loc\}) \quad in\rightarrow\sigma'(loc)}{\sigma'=e(\sigma)}\,S$$

$$\frac{e=(RS,(in_1,in_2),(out),\{loc\}) \quad in_1\wedge\neg in_2\rightarrow\neg\sigma'(loc) \quad in_2\rightarrow\sigma'(loc) \quad \sigma'(out)=\sigma'(loc)}{\sigma'=e(\sigma)}\,RS$$

$$\frac{e=(SR,(in_1,in_2),(out),\{loc\}) \quad \neg in_1\wedge in_2\rightarrow\sigma'(loc) \quad in_1\rightarrow\neg\sigma'(loc) \quad \sigma'(out)=\sigma'(loc)}{\sigma'=e(\sigma)}\,SR$$

$$\frac{e=(N,(in),(out),\{loc\}) \quad \sigma'(loc)=\sigma(in) \quad \sigma'(out)=\sigma(loc)\wedge\neg\sigma(in)}{\sigma'=e(\sigma)}\,N$$

$$\frac{e=(P,(in),(out),\{loc\}) \quad \sigma'(loc)=\sigma(in) \quad \sigma'(out)=\neg\sigma(loc)\wedge\sigma(in)}{\sigma'=e(\sigma)}\,P$$

Vergleicher

$$\frac{e=(CMP==I,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=(\sigma(in_1)=\sigma(in_2))}{\sigma'=e(\sigma)}\,CMP == I$$

$$\frac{e=(CMP<>I,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=\neg(\sigma(in_1)=\sigma(in_2))}{\sigma'=e(\sigma)}\,CMP <> I$$

$$\frac{e=(CMP>I,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=(\sigma(in_1)>\sigma(in_2))}{\sigma'=e(\sigma)}\,CMP > I$$

$$\frac{e=(CMP<I,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=(\sigma(in_1)<\sigma(in_2))}{\sigma'=e(\sigma)}\,CMP < I$$

$$\frac{e=(CMP>=I,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=((\sigma(in_1)>\sigma(in_2))\vee(\sigma(in_1)=\sigma(in_2)))}{\sigma'=e(\sigma)}\,CMP >= I$$

$$\frac{e=(CMP<=I,(in_1,in2),(out),\varnothing) \quad \sigma'(out)=((\sigma(in_1)<\sigma(in_2))\vee(\sigma(in_1)=\sigma(in_2)))}{\sigma'=e(\sigma)}\,CMP <= I$$

Tabelle 4.2: Operationale Regeln

Eigenschaften der Programmausführung

Wie in Kapitel 2 ausführlich beschrieben, ist die zyklische Verarbeitung der in SPS gespeicherten Programme ein wichtiges Merkmal von SPS. Am Anfang eines Zyklus' (Taktes) werden die Eingänge gelesen. Nach dieser Phase wird das Programm ausgeführt und am Ende des Zyklus' werden die Ausgänge geschrieben. Danach fängt ein neuer Zyklus an. Durch die komplette Bearbeitung eines Programms wird ein SPS-Zustandsübergang definiert.

Die Verarbeitung eines FUP-Programms wird im Folgenden durch mehrere sukzessive Schritte betrachtet. Dabei steht ein einzelner Schritt für die Veränderung des Programmzustands nach der Ausführung eines Elementes innerhalb eines FUP-Netzwerks. Die Übergänge sollten dann zusammengefasst werden, um die Ausführung eines Programms als Zustandsübergang zu repräsentieren.

Wie bereits erwähnt, wird das Verhalten des betrachteten Programms sequentiell bezüglich der Ausführungsreihenfolge der Elemente im Netzwerk beschrieben. Bei der Ausführung eines Netzwerks N bestimmen die Verbindungen zwischen Elementen die Reihenfolge der Ausführung der Elemente. Dabei lässt sich aus der Menge der noch nicht ausgeführten Elementen $E \subset N$ das nächste auszuführende Element eindeutig bestimmen. Die Vorgehensweise lässt sich durch eine partielle Abbildung $next : 2^N \rightharpoonup N$ beschreiben. Bevor diese Abbildung definiert wird, werden zwei wichtige Eigenschaften eines FUP-Programms in den folgenden Lemmata zusammengefasst.

Lemma 4.1. *Sei E die Menge der noch nicht ausgeführten Elemente des FUP-Netzwerks N. Sei $F \subset E$ die Menge der Elemente e mit der Eigenschaft, dass jeder Eingang in^e in $V \cup D$ liegt oder es ein bereits ausgeführtes Element $e' \in N \setminus E$ mit $out^{e'} = in^e$ gibt. Wenn es in F zwei Elemente e' und e'' ohne Ausgänge gibt, sind die Eingänge dieser Elemente die Ausgänge eines Elements aus $N \setminus E$.*

Beweis. Setzen wir voraus, dass es zwei verschiedene Elemente g' und g'' aus $N \setminus E$ mit folgenden Eigenschaften gibt: $out^{g'} = in^{e'}$ und $out^{g''} = in^{e''}$. Bei der Programmausführung können nicht zwei Elemente gleichzeitig ausgeführt werden. Setzen wir voraus, dass g' vor g'' ausgeführt wurde. Nach seiner Ausführung wird auch e' ausführungsbereit. Da e' keine Ausgänge hat, sollte e' vor g'' ausgeführt werden. Das widerspricht der Voraussetzung, dass e' der Menge der noch nicht ausgeführten Elementen gehört, $e' \in E$, und dass g'' bereits ausgeführt wurde, $g'' \in N \setminus E$. □

Lemma 4.2. *Sei E die Menge der noch nicht ausgeführten Elemente des FUP-Netzwerks N. Seien e' und e'' zwei Elemente aus E, die Ausgänge haben und deren Eingänge entweder $V \cup D$ angehören, oder zu deren Eingängen es Leitungen von bereits ausgeführten Elementen gibt. Es existiert ein $e \in E$, so dass $e' \prec e$, $e'' \prec e$ und e' und e'' führen zu verschiedenen Eingängen von e.*

Beweis. Nach Definition 4.2 gibt es eine Reihe von Elementen $e_1, \ldots, e_k \in E$, so dass $e' = e_1$, $e'' = e_k$, und

$$(\forall i < k)(\exists \_L_i \in W) \_L_i = out^{e_i} = in^{e_{i+1}} \veebar \_L_i = in^{e_i} = out^{e_{i+1}}$$

Das heißt, für jedes Element e_i, $1 < i < k$, gilt entweder $out^{e_i} = in^{e_{i+1}}$ oder $in^{e_i} = out^{e_{i+1}}$. Sei e das Element mit dem kleinsten Index j mit der Eigenschaft $in^{e_j} = out^{e_{j+1}}$. Dann gilt Folgendes:

- e existiert:
  Gehen wir vom Gegenteil aus. Dann gilt $out^{e_i} = in^{e_{i+1}}$ für jedes Element e_i. Dementsprechend würde das auch für e_{k-1} gelten, im Widerspruch zur Voraussetzung $out^{e_k} = in^{k-1}$.

- e ist das gesuchte Element:
  Aus $out^{e_i} = in^{e_{i+1}}$, $\forall i$ $1 < i < j$, und aus der Definition der Elemente $e_1, \ldots, e_k \in E$ folgt $e_1 \prec e_j$ ($e' \prec e$). Zusätzlich folgt aus $in^{e_j} = out^{e_{j+1}}$ $e_k \prec e_j$ ($e'' \prec e$). Dabei führen e' und e'' zu verschiedenen Eingängen von e, da laut Definition 4.2 der Datenfluss eine $1-n$ Relation ist: ein Ausgang kann zu verschiedenen Eingängen führen, verschiedene Ausgänge können aber nicht zum gleichen Eingang führen.

□

Die Abbildung $next\colon 2^N \rightharpoonup N$, die Ausführungsreihenfolge der Elemente im Netzwerk definiert, lässt sich folgendermaßen definieren:

$$next(E) = e' \text{ genau dann wenn } e' \text{ Folgendes erfüllt:}$$

1. Für jeden Eingang in von e' gilt $in \in V \cup D$ oder es gibt einen Ausgang out von einem Element aus $N \setminus E$ so dass $in = out$.

2. Falls es eine Menge $F \subset N$ gibt, deren Elemente die vorherige Eigenschaft erfüllen, dann gilt Folgendes:

   (a) Falls es in F ein Element gibt, das keine Ausgänge hat, dann ist dies e'.

   (b) Wenn es mehrere Elemente ohne Ausgänge gibt, dann sind dies Elemente, zu denen Ausgänge eines Elements f aus $N \setminus E$ führen (Lemma 4.1). Der Ausgang von f mit dem kleinsten Index führt dann zu e'. Wenn der Ausgang mehrere Abzweige hat, dann führt der Ausgang mit der kleinsten Abzweigsnummer zu e'.

   (c) Falls keine der vorherigen zwei Situationen auftritt, gilt laut Lemma 4.2 Folgendes: für e' und ein beliebiges $e'' \in F$ existiert ein $e \in E$, so dass $e' \prec e$, $e'' \prec e$ und e' und e'' führen zu verschiedenen Eingängen von e. Dann ist e' das Element, das zum Eingang mit dem kleineren Index führt.

3. Wenn kein Element vorhanden ist, das die obigen Eigenschaften erfüllt, dann ist e' nicht definiert.

Operationale Semantik

Um das Verhalten des Programms zu beschreiben, wird nun der Begriff Konfiguration eingeführt.

Definition 4.5. *(Konfiguration)*
Die Konfiguration eines Netzwerks N ist ein Tupel $c = (\sigma, e, E)$, wobei

- *$\sigma \in \Sigma$ ein Zustand der Programmvariablen ist,*
- *$E \subset N$ die Menge der noch nicht ausgeführten Bausteinelementen ist und*
- *$e \in N \setminus E$ dasjenige Element im Netzwerk ist, das als nächstes ausgeführt werden soll.*

Die Semantik eines FUP-Programms lässt sich mit Hilfe eines Transitionssystems beschreiben, das folgendermaßen definiert wird.

Definition 4.6. *(FUP-Transitionssystem)*
Jedem FUP-Netzwerk N kann ein Transitionssystem $\mathcal{T} = (\mathcal{C}, c_0, \rightsquigarrow)$ zugewiesen werden, wobei

- *$\mathcal{C}$ die Menge der Konfigurationen ist,*

- $c_0 = (\sigma_0, e_0, E_0)$ *die Startkonfiguration ist, wobei*
  - σ_0 *der Startzustand ist, in dem alle Variablen ihren Anfangswerten haben,*
  - $e_0 = next(N)$ *das Startelement ist, und*
  - $E_0 = N \setminus \{e_0\}$
- $\rightsquigarrow$ *ist eine Transitionsrelation, die folgendermaßen definiert ist: für zwei beliebige Konfigurationen* $c = (\sigma, e, E)$ *und* $c' = (\sigma', e', E')$ *gilt* $c \rightsquigarrow c'$ *genau dann, wenn*

$$\sigma' = e(\sigma) \ \wedge \ e' = next(E) \ \wedge \ E' = E \setminus \{e\}.$$

Definition 4.7. *(Ausführung von $\mathcal{T}$)*
Eine Ausführung von $\mathcal{T}$ ist eine endliche oder unendliche Reihe $\langle c_1, c_2, \ldots \rangle$, wobei $c_i \rightsquigarrow c_{i+1}$, für jedes i.

Definition 4.8. *(Operationale Semantik)*
Die operationale Semantik $[\![\mathcal{P}]\!]$ eines FUP-Programms $\mathcal{P}$ ist die Menge der Ausführungen des Transitionssystems $\mathcal{T}(\mathcal{P})$.

4.2.4 Beispiel

In diesem Abschnitt werden an einem Beispiel die bereits eingeführten Begriffe vorgestellt. In Abbildung 4.3 ist das FUP-Netzwerk $\mathcal{N} = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ gegeben, wobei:

$$\begin{aligned}
e_1 &= \{CMP{==}I, (int1, 20), (_L1), \varnothing\}, \\
e_2 &= \{\&, (_L1, bool1, bool2), (_L2), \varnothing\}, \\
e_3 &= \{\&, (bool3, bool4), (_L3), \varnothing\}, \\
e_4 &= \{>= 1, (_L2, _L3), (_L4), \varnothing\}, \\
e_5 &= \{=, (_L4), \varnothing, \{result1\}\}, \\
e_6 &= \{=, (_L4), \varnothing, \{result2\}\}.
\end{aligned}$$

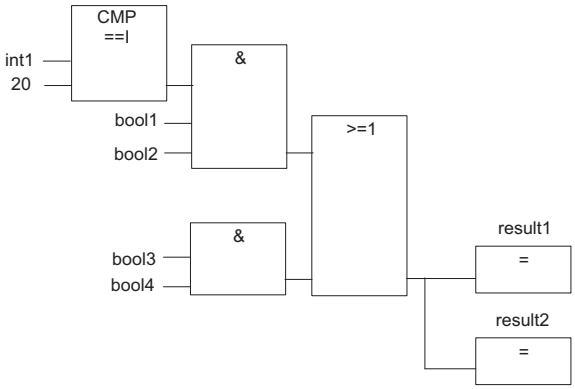


Abbildung 4.3: FUP-Beispiel

5 Model Checking von FUP-Programmen

Unabhängig von der Entstehung neuer theoretischer Erkenntnisse im Gebiet der formalen Verifikation werden die technischen Fähigkeiten von Rechnern ständig verbessert. Dies führt dazu, dass man mit den noch vor zehn Jahren entwickelten Verfahren heute deutlich grössere Programme verifizieren kann als damals. Jedoch gibt es immer noch kein standardisiertes Vorgehen zur SPS-Verifikation. Dieses Kapitel stellt ein Verfahren für die Verifikation von SPS-Software vor, die in der Programmiersprache FUP geschrieben wurde.

Die Verifikation wird mit Hilfe von Model Checking durchgeführt. Als Verifikationstool wird der Model Checker NuSMV (a New Symbolic Model Verifier) benutzt. Wie bereits im vorherigen Kapitel in Abschnitt 3.3 über Model Checking beschrieben, gibt es beim Model Checking zwei wichtige Schritte:

- Erstellung eines NuSMV-Modells; und
- Darstellung der zu testenden Spezifikation mittels temporaler Logik.

Bei dem hier entwickelten Verfahren wird ein NuSMV-Modell in drei Schritten erstellt.

1. Zuerst wird das Programm in einem textuellen Format dargestellt (textFUP), in dem jeder graphische FUP-Operator durch eine textuelle Anweisung repräsentiert ist.
2. Danach wird eine Vereinfachung des textFUP-Formats vorgenommen. Dies wird im tFUP-Format gespeichert.
3. Anschließend wird aus der tFUP-Darstellung das entsprechende NuSMV-Modell erstellt.

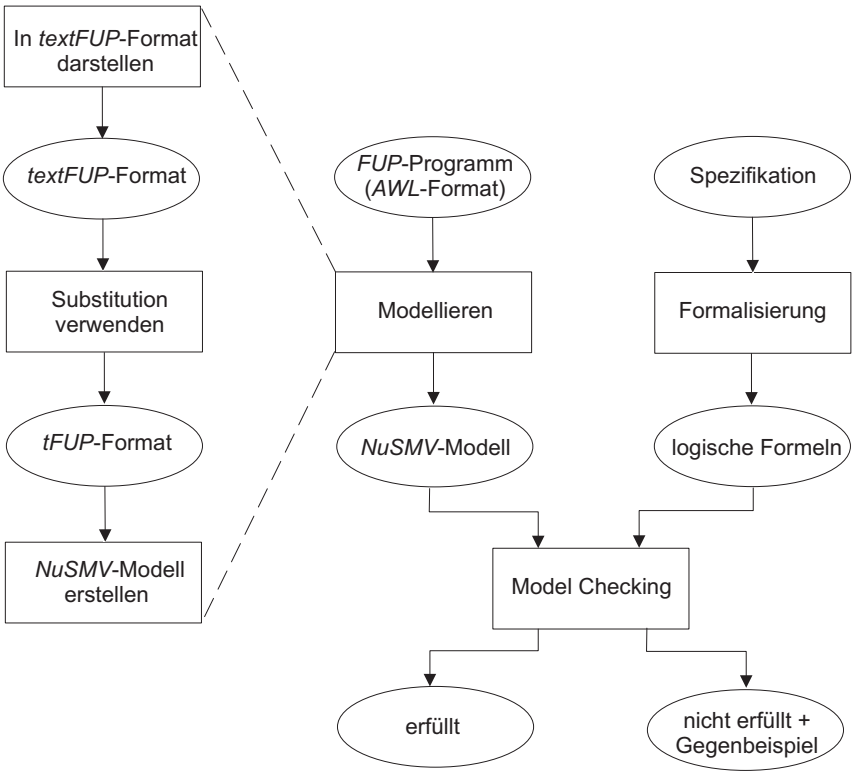


Abbildung 5.1: Model Checking von FUP-Programmen

Eine Übersicht über das ganze Verfahren wird in Abbildung 5.1 gezeigt.

5.1 Motivation

Bevor das Verfahren für die FUP-Verifikation im Detail vorgestellt wird, wird es in diesem Abschnitt eine kurze Motivation dafür geben. Das Hauptziel der vorliegenden Arbeit ist es, einen Beitrag für die formale Verifikation von SPS-Software zu leisten. Obwohl SPS-Software häufig in FUP geschrieben wird, lässt sich eine textuelle Darstellung viel leichter untersuchen als eine graphische Darstellung des Programms. Da die AWL-Darstellung eines FUP-Programms jederzeit zur Verfügung steht, wurde der Schwerpunkt in der ersten Phase dieses Projektes auf die Verifikation von AWL-Software gesetzt. Eine kurze Erklärung der Vorgehensweise für die AWL-Verifikation wird im folgenden Abschnitt vorgestellt. Die durch AWL-Verifikation gewonnene Erfahrung stellt gleichzeitig eine Motivation für die FUP-Verifikation dar.

AWL-Verifikation

Die maschinenorientierte SPS-Programmiersprache AWL wurde in Kapitel 2 eingeführt. Das folgende Verfahren für die AWL-Verifikation wurde in [PPKE07] veröffentlicht. Die Automatisierung des Verfahrens wurde in [PPK07] beschrieben. Die Grundprinzipien des Verfahrens werden im Folgenden beschrieben, indem das formale Modell eines AWL-Programms vorgestellt wird. Die Überführung dieses Modells beispielsweise in ein NuSMV-Modell folgt dann ähnlich wie die Überführung des tFUP-Modells in ein NuSMV-Modell, was im Laufe des Kapitels erklärt wird.

Das Hauptmerkmal der Modellierung eines AWL-Programms ist, dass die hardwarespezifischen Eigenschaften der entsprechenden SPS in der Modellbeschreibung enthalten sind. Analog zu einem textFUP- oder einem tFUP-Programm kann ein AWL-Programm durch ein Transitionssystem, das ähnlich dem in Definition 5.2 vorgestellten Transitionssystem ist, beschrieben werden. Der Hauptunterschied liegt allerdings in den Definitionen der Konfigurationen, durch die die Zustände in den Transitionssystemen definiert sind.

AWL-Konfiguration eines Netzwerks

Im Folgenden wird die AWL-Darstellung eines FUP-Netzwerks als ein AWL-Netzwerk betrachtet. Das formale Modell eines AWL-Netzwerks kann durch ein Transitionssystem dargestellt werden, dessen Zustandsmenge aus AWL-Konfigurationen besteht. Die AWL-Konfiguration kann als ein Tupel (σ, a, pc) betrachtet werden, wobei σ eine Belegung der Variablen ist, a die AWL-Anweisung, die als nächste ausgeführt werden soll und pc die Programmzeile, in der sich die Anweisung a im Programm befindet.

Im Vergleich zur späteren Definition eines tFUP-Netzwerks gibt es zwei wichtige Punkte, die eine AWL-Konfiguration komplexer machen

- Programmzähler - Da mehrere AWL-Anweisungen durch eine textFUP-Anweisung (ein FUP-Element) dargestellt werden, hat ein AWL-Netzwerk eine vielfache Anzahl an Codezeilen im Vergleich zum entsprechenden textFUP-Netzwerk. Als Beispiel kann das in Abbildung 4.3 gegebene Netzwerk betrachtet werden. Im Vergleich zu 12 Codezeilen in der AWL-Darstellung des Netzwerks, gibt es in der tFUP-Darstellung nur 2 Zeilen (siehe Abbildung 5.2).

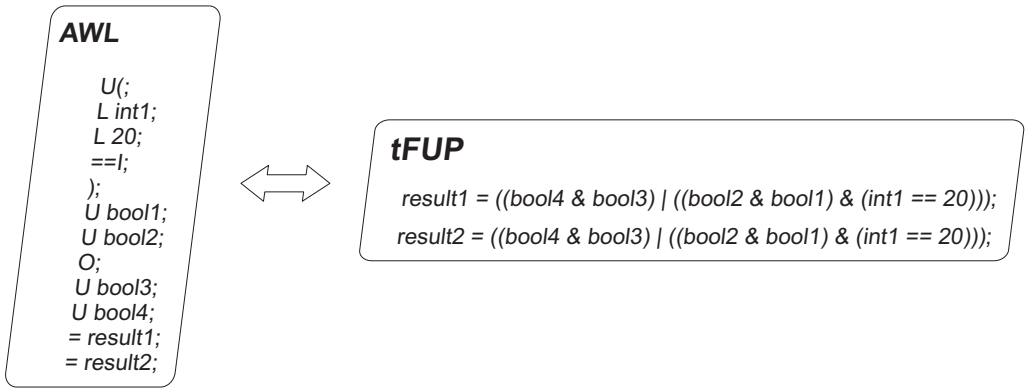


Abbildung 5.2: AWL- und tFUP-Format eines FUP-Netzwerks

- Variablen - Im formalen Modell eines AWL-Netzwerks werden sowohl software- als auch hardwarespezifische Variablen betrachtet. Zu den softwarespezifischen Variablen gehören alle Variablen, die im Programm deklariert sind. Im Gegensatz zu textFUP oder tFUP reichen diese Variablen nicht, um die Semantik der AWL-Anweisungen vollständig zu beschreiben. Daher werden hardwarespezifische Variablen für die Modellierung von Hardwareaspekten, wie Akkumulatoren, Statusbits oder Klammerstack, benutzt. Diese Begriffe wurden bereits in Kapitel 2 eingeführt.

Erfahrung in der AWL-Verifikation

Als Fallstudie für das vorgestellte Verfahren der AWL-Verifikation wurde die gleiche Softwarekomponente benutzt, die bei der FUP-Verifikation eingesetzt wurde (siehe Kapitel 6). Allerdings konnten mit dem Verfahren nur einzelne Funktionen verifiziert werden, die im Hauptprogramm aufgerufen werden. Die ganze Softwarekomponente konnte man wegen der Modellgröße mit dem Verfahren nicht verifizieren.

Das vorgestellte Verfahren war in der Lage auf jedes AWL-Programm angewendet zu werden. Allerdings war gedacht, das Verfahren auf ein FUP-Programm anzuwenden. Die AWL-Repräsentation eines FUP-Programms hat sich nicht als eine geeignete Form für die FUP-Verifikation gezeigt. Daher wurde die folgende Frage gestellt:

„Kann man ein FUP-Programm in einem textuellen Format darstellen, das für Model Checking geeignet ist und in dem hardwarespezifische Eigenschaften einer SPS nicht sichtbar sind?“

In dieser Frage liegt die Motivation für das Verfahren für die FUP-Verifikation,

das im Folgenden vorgestellt wird. Als Antwort auf die Frage wurden das textFUP- und das tFUP-Format eines FUP-Programms entwickelt, die zunächst beschrieben werden.

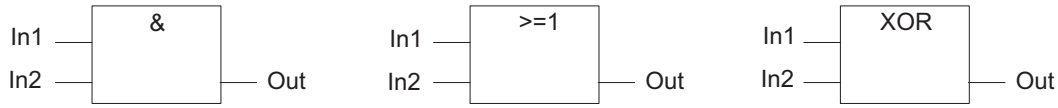
5.2 textFUP - Textuelle Darstellung von FUP

Um ein FUP-Programm zu verifizieren, wird das Programm zuerst in einem textuellen Format dargestellt (textFUP-Format), auf welches formale Verifikation effizient angewendet werden kann. Ein Teil der textFUP-Anweisungen wird detailliert im folgenden Abschnitt vorgestellt. Danach wird die Semantik von textFUP formal beschrieben. Anschließend wird die Äquivalenz zwischen FUP und textFUP gezeigt.

5.2.1 Syntax von textFUP

Im textFUP-Format eines FUP-Programms wird jeder graphische FUP-Operator durch eine textuelle Repräsentation dargestellt. Neben der Beschreibung von textFUP bietet dieser Abschnitt auch einen tieferen Einblick in die FUP-Sprache. Dazu werden im Folgenden textFUP-Anweisungen analog zur Klassifizierung von FUP-Operatoren in Tabelle 4.1 in Abschnitt 4 eingeführt.

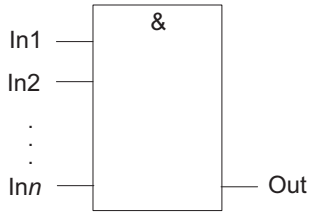
- Bitverknüpfungsoperationen
  Logische *UND-*, *ODER-* und *Exklusiv-ODER*-Operationen:



  werden in textFUP folgendermaßen dargestellt

$$Out = (In1 \,\&\, In2)$$
$$Out = (In1 \mid In2)$$
$$Out = (In1 \; XOR \; In2)$$

  Die Operatoren *UND* und *ODER* können in FUP mehr als zwei Eingänge haben, wie z.B.



Das entsprechende textFUP-Konstrukt dafür ist

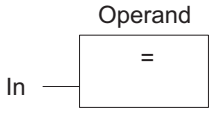
$$Out = ((In1 \ \& \ In2) \ \& \ \ldots \ \& \ Inn)$$

Die Eingänge werden wie hier geklammert dargestellt, um auch den Fall fehlender Assoziativität des Operators semantisch eindeutig zu definieren.

Ein binärer Eingang wird in FUP folgendermaßen negiert

Dies wird in textFUP durch $!In$ dargestellt.

Die FUP-Zuweisung



wird einfach durch $Operand = In$ repräsentiert.

Zu den Bitverknüpfungsoperationen gehören auch die Operationen *Ausgang rücksetzen* (R) und *Ausgang setzen* (S).

Falls am Eingang des R-Operators *true* vorliegt, wird der Operand auf *false* gesetzt. Wenn am Eingang *false* anliegt, bleibt der Operand unverändert. Beim Operator S wird der Operand nur dann gesetzt, wenn am Eingang *true* anliegt. Die genauere Semantik der Operatoren ist in Abschnitt 4 in Tabelle 4.2 dargestellt. Nun soll aber ihre Syntax weiter betrachtet werden. Die Operatoren werden in textFUP wie folgt dargestellt

$$R(Operand, \ In)$$

$$S(Operand, \ In)$$

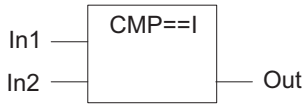
Mit dem N-Operator (*Flanke 1→0 abfragen*) wird der Signalzustand am Eingang mit dem Signalzustand im Operanden (Flankenmerker) verglichen. Falls

am Eingang ein *false* anliegt und im Operanden im vorherigen Zyklus *true* gespeichert wurde, wird eine *fallende Flanke* erkannt. In dem Fall wird der Ausgang auf *true* gesetzt, sonst auf *false*. Der *P*-Operator (*Flanke 0⟶1 abfragen*), mit dem eine *steigende Flanke* erkannt wird, funktioniert genau umgekehrt. Diese Operatoren

werden in textFUP mit folgenden Ausdrücken ersetzt:

$$Out = N(Operand,\ In)$$
$$Out = P(Operand,\ In)$$

- Vergleicher Um zwei Eingänge zu vergleichen, kann man folgende Vergleichsarten verwenden: gleich, ungleich, größer, größer oder gleich, kleiner sowie kleiner oder gleich. Beispielsweise wird der Operator,



der zwei Integer-Eingänge auf Gleichheit überprüft, in textFUP durch $Out = (In1\ ==\ In2)$ dargestellt.

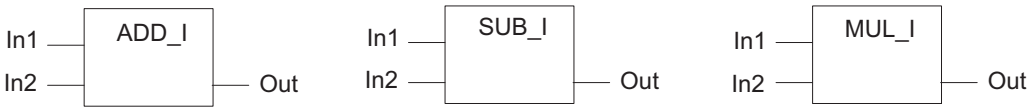
- Sprünge
  Bei Sprungoperationen wird zwischen bedingten und absoluten Sprüngen unterschieden. Wenn am Eingang ein *true* vorliegt, kann ein bedingter Sprung durch *JMP* realisiert werden. Damit springt man im Programm auf die mit *Label* markierte Stelle. Wenn am Eingang ein *false* vorliegt, kann der Sprung durch *JMPN* durchgeführt werden.

  Dies wird in textFUP wie folgt repräsentiert:

$$JMP(In,\ Label)$$
$$JMPN(In,\ Label)$$

  Ein absoluter Sprung entspricht einer *goto*-Anweisung und wird einfach durch $JMP(true,\ Label)$ dargestellt.

- Ganzzahl-Arithmetik



  Addition, Subtraktion oder Multiplikation von zwei Integer-Werten $In1$, $In2$ wird in textFUP folgendermaßen ausgedrückt:

$$ADD\_I(Out,\ In1,\ In2)$$
$$SUB\_I(Out,\ In1,\ In2)$$
$$MUL\_I(Out,\ In1,\ In2)$$

- Wertzuweisung
  Mit dem *MOVE*-Operator wird der Wert, der als Eingang anliegt, in den Operanden kopiert, der als Ausgang angegeben ist: $Out = In$.

Um einen besseren Einblick in textFUP zu bieten, wird später in Abbildung 5.6 das textFUP-Format des in Kapitel 4 eingeführten Beispiels (Beispiel 4.3) gezeigt.

5.2.2 textFUP-Semantik

Die FUP-Semantik wurde bereits in Kapitel 4 vorgestellt. In diesem Abschnitt wird die Semantik der Sprache aus der Sicht von textFUP beschrieben. Anschließend wird eine Parallele zwischen beiden Beschreibungen gezogen. Dazu werden, analog zu den FUP-Begriffen in Kapitel 4, zuerst einige grundlegende textFUP-Begriffe eingeführt.

Ein FUP-Baustein (siehe Definiton 4.1) wird in textFUP einfach durch die entsprechende textFUP-Anweisung gemäß des vorherigen Abschnitts ersetzt. Dafür kann eine Abbildung $\bar{h}$: $e \mapsto a$ definiert werden, die jedem FUP-Element e die entsprechende textFUP-Repräsentation a zuweist. Die Reihenfolge, in der FUP-Operatoren in einem Netzwerk ausgeführt werden, ist bei der Beschreibung der FUP-Semantik durch die Abbildung *next* bestimmt. Diese Rolle übernimmt bei textFUP der Programmzähler, der als eine Abbildung p: $a \mapsto pc$, $pc \in \mathbb{N}$, definiert wird, die jeder Anweisung a ihre Zeilennummer pc im Programm zuweist. Diese Abbildung ist folgendermaßen definiert:

- Wenn a die erste textFUP-Anweisung im Programm ist, dann ist $p(a) = 1$.
- Andernfalls wird dasjenige Element e betrachtet, für das $a = \bar{h}(e)$ gilt. Da e nicht als erstes Element im Programm ausgeführt wird, hat das Element

folgende Eigenschaft: Es gibt ein Element e' und eine Menge E' von Elementen, die nach e' ausgeführt werden, so dass $e = next(E')$. Sei $a' = \bar{h}(e')$, dann ist $p(a) = p(a') + 1$.

Wie bereits gesagt, wird mit dieser Definition die Reihenfolge bestimmt, in der die FUP-Operatoren ausgeführt werden. Dabei ist besonders wichtig für die Berücksichtigung von Sprunganweisungen, dass die ganzen Betrachtungen innerhalb eines Netzwerks gemacht sind. Ein Netzwerk im FUP-Programm ist eine maximale Menge von grafisch miteinander verknüpften Elementen. Eine Sprunganweisung hat keine Ausgänge. Das heißt, dass nach einer Sprunganweisung keine Verknüpfungen möglich sind. Damit werden auch Sprunganweisungen mit der Definition von der Abbildung *next* umgefasst.

Ein Netzwerk N (siehe Definition 4.2) mit n Bausteinen lässt sich dann als eine Reihe von textFUP-Anweisungen $\{a_i \mid m \leq i \leq m+n\}$ darstellen, wobei $m \in \mathcal{N}$ die Programmzeile ist, ab der die Beschreibung des Netzwerks N im Programm beginnt.

Definition 5.1. *(textFUP-Konfiguration)*
Die textFUP-Konfiguration eines Netzwerks N ist ein Tupel $d = (\sigma, a, pc)$, wobei

- *$\sigma \in \Sigma$ ein Zustand der Programmvariablen ist,*
- *a die textFUP-Anweisung ist, die als nächste ausgeführt werden soll und*
- *$pc \in N$ die Programmzeile ist, in der sich die Anweisung a im Programm befindet ($pc = p(a)$).*

Definition 5.2. *(textFUP-Transitionssystem)*
Jedem textFUP-Netzwerk kann ein Transitionssystem $\mathcal{S} = (\mathcal{D}, d_0, \hookrightarrow)$ zugewiesen werden, wobei

- *$\mathcal{D}$ die Menge der textFUP-Konfigurationen ist,*
- *$d_0 = (\sigma_0, a_0, 1)$ die Startkonfiguration mit dem Anfangszustand der Programmvariablen σ_0 und mit der ersten Anweisung a_0 ist.*
- *$\hookrightarrow$ eine Transitionsrelation ist, die folgendermaßen definiert ist: Für zwei Konfigurationen $d = (\sigma, a, pc)$ und $d' = (\sigma', a', pc')$ gilt $d \hookrightarrow d'$ genau dann, wenn $\sigma' = a(\sigma)$ und $pc' = pc + 1$.*

Analog zur Definition 4.7 und Definition 4.8 in Kapitel 4 kann eine Ausführung von $\mathcal{S}$ und die operationale Semantik eines textFUP-Programms definiert werden.

Definition 5.3. *(Ausführung von $\mathcal{S}$)*
Eine Ausführung von $\mathcal{S}$ ist eine endliche oder unendliche Reihe $\langle d_1, d_2, \ldots \rangle$, wobei $d_i \hookrightarrow d_{i+1}$, für jedes i.

Definition 5.4. *(Operationale Semantik)*
Die operationale Semantik $[\![\mathcal{P}]\!]$ eines textFUP-Programms $\mathcal{P}$ ist die Menge der Ausführungen des Transitionssystems $\mathcal{S}(\mathcal{P})$.

5.2.3 Isomorphie

Es gibt verschiedene Äquivalenzrelationen, die zwischen zwei Transitionssystemen existieren können: Isomorphie, starke Bisimulation, Beobachtungskongruenz, schwache Bisimulation, divergente Bisimulation, usw. Einen Überblick über alle diese Relationen findet man in [Chr95]. Die Definition von Isomorphie wird dabei aus [Tau89] übernommen. Dementsprechend lässt sich die Isomorphie von Transitionssystemen eines FUP- und eines textFUP-Netzwerks folgendermaßen definieren.

Definition 5.5. *(Isomorphie)*
Seien N ein FUP-Netzwerk, $\mathcal{T} = (\mathcal{C}, c_0, \rightsquigarrow)$ das entsprechende FUP-Transitionssystem und $\mathcal{S} = (\mathcal{D}, d_0, \hookrightarrow)$ das entsprechende textFUP-Transitionssystem. $\mathcal{T}$ und $\mathcal{S}$ heißen isomorph ($\mathcal{T} \cong \mathcal{S}$), wenn eine Abbildung $h\colon \mathcal{C} \to \mathcal{D}$ mit den folgenden Eigenschaften existiert:

1. *h ist bijektive,*
2. *$h(c_0) = d_0$ und*
3. *für alle $c, c' \in \mathcal{C} \quad c \rightsquigarrow c' \Leftrightarrow h(c) \hookrightarrow h(c')$ gilt.*

Lemma 5.1. *Seien N ein FUP-Netzwerk, $\mathcal{T} = (\mathcal{C}, c_0, \rightsquigarrow)$ das entsprechende FUP-Transitionssystem und $\mathcal{S} = (\mathcal{D}, d_0, \hookrightarrow)$ das entsprechende textFUP-Transitionssystem, wobei $c_0 = (\sigma_0, e_0, E_0)$ und $d_0 = (\sigma_0, a_0, 1)$. Sei $h\colon (\sigma, e, E) \mapsto (\sigma, a, pc)$ eine Abbildung, die jeder Konfiguration aus $\mathcal{C}$ eine Konfiguration aus $\mathcal{D}$ zuordnet, wobei $a = \bar{h}(e)$ und $pc = p(a)$. Dann ist h ein Isomorphismus.*

Beweis. Damit $\mathcal{T}$ und $\mathcal{S}$ isomorph sind, muss h folgende Bedingungen erfüllen

1. h ist eine Bijektion

2. $h(c_0) = d_0$

3. $c \rightsquigarrow c' \quad \Leftrightarrow \quad h(c) \hookrightarrow h(c')$

Das kann man folgendermaßen zeigen.

1. h ist eine Bijektion

   - h ist *injektiv*: $h(c) = h(c') \Rightarrow c = c'$
     Seien $c = (\sigma, e, E)$ und $c' = (\sigma', e', E')$ zwei beliebige Konfigurationen aus $\mathcal{C}$. Dann gilt für $h(c) = (\sigma, a, pc)$ und $h(c') = (\sigma', a', pc')$ Folgendes:

     $$h(c) = h(c') \;\Leftrightarrow\; (\sigma, a, pc) = (\sigma', a', pc') \;\Leftrightarrow\; \sigma = \sigma' \wedge a = a' \wedge pc = pc'.$$

     Dann muss auch $e = e'$ und $E = E'$ gelten, weil in einem beliebigen Moment nur ein FUP-Operator ausgeführt werden kann. Das heißt, es kann nicht sein, dass es zwei verschiedene FUP-Elemente e und e' gibt, die in eine Anweisung a auf einer Programmzeile pc abgebildet werden. Also muss $c = c'$ gelten.
   - h ist *surjektiv*: $(\forall d \in \mathcal{D})\ (\exists c \in \mathcal{C})\ h(c) = d$
     Dies gilt für h per Konstruktion: Jede Anweisung a in einer Konfiguration $(\sigma, a, pc) \in \mathcal{D}$ ist eine textFUP-Darstellung eines FUP-Operators e. Wenn man mit E die Menge der FUP-Operatoren bezeichnet, die nach e ausgeführt werden, ist dann (σ, e, E) die gesuchte Konfiguration.

2. $h(c_0) = d_0$
   Dies gilt per Definition: $h(c_0) = h(\sigma_0, e_0, E_0) = (\sigma_0, a_0, 1) = d_0$

3. $c \rightsquigarrow c' \quad \Leftrightarrow \quad h(c) \hookrightarrow h(c')$
   Sei $c = (\sigma, e, E)$, $c' = (\sigma', e', E')$, $h(c) = (\sigma, a, pc)$ und $h(c') = (\sigma', a', pc')$. Dann gilt:

   $$c \rightsquigarrow c' \quad \Leftrightarrow \quad \sigma' = e(\sigma) \wedge e' = next(E) \wedge E' = E \setminus \{e\}$$

   Diese Aussage ist nach Definition des Programmzählers äquivalent mit folgender Aussage:

   $$\sigma' = a(\sigma) \;\wedge\; p(a') = p(a) + 1 \quad \Leftrightarrow \quad \sigma' = a(\sigma) \;\wedge\; pc' = pc + 1 \quad \Leftrightarrow$$
   $$(\sigma, a, pc) \hookrightarrow (\sigma', a', pc') \quad \Leftrightarrow \quad h(c) \hookrightarrow h(c')$$

□

5.3 tFUP - Substitution von textFUP

Im vorherigen Abschnitt wurde gezeigt, dass FUP- und textFUP-Transitionssysteme äquivalent sind. Die Sprache vom Model Checker NuSMV ist so konzipiert, dass sich damit Zustandsautomaten effizient beschreiben lassen. Intuitiv ist es leichter, das textFUP-Transitionssystem in NuSMV darzustellen. Bevor das textFUP-Transitionssystem in der Sprache von NuSMV dargestellt wird, kann eine Vereinfachung des textFUP-Programms vorgenommen werden. Das neue Format, das tFUP genannt wird, hat die Eigenschaft, dass in ihm die Anzahl der Leitungsvariablen minimiert wird. Eine textFUP-Zeile, in der eine neue Leitungsvariable entsteht, kann in tFUP unter bestimmten Bedingungen ausgelassen werden. Dazu wird analysiert, welcher Ausdruck einer bestimmten Variablen zugewiesen wird. Dabei merkt man welcher Term der Variable zugewiesen werden soll. Dann wird in jeder anderen Zeile, in der die Leitungsvariable verwendet wird, eine Substitution der Variable durch den entsprechenden Term vorgenommen.

Um diese Vorgehensweise anschaulicher zu machen, wird in Abbildung 5.6 das tFUP-Format des erwähnten Beispiels angegeben.

Einem tFUP-Netzwerk kann ein Transitionssystem zugewiesen werden, dass in Definition 5.2 definiert ist. Es muss nur noch festgestellt werden, inwiefern diese textFUP- und tFUP-Darstellung eines FUP-Netzwerks voneinander abhängig sind. Wie im vorherigen Abschnitt erwähnt, ist Isomorphie die stärkste Äquivalenzrelation zwischen zwei Systemen. Die konnte zwischen FUP und tFUP nachgewiesen werden, was für tFUP und textFUP nicht der Fall ist. Im Allgemeinen sind die zwei Systeme nicht isomorph. Allerdings kann zwischen diesen Systemen eine starke Bisimulation erkannt werden. Dies wird im Folgenden genauer betrachtet.

textFUP- und tFUP-Transitionssystem im Vergleich

Sei N ein FUP-Netzwerk und $\mathcal{S} = (\mathcal{D}, d_0, \hookrightarrow)$ das entsprechende textFUP-Transitionssystem. Sei $\mathcal{S}' = (\mathcal{D}', d_0', \hookrightarrow')$ ein Transitionssystem, das durch folgende Regeln eingeführt wird.

- $\mathcal{D}'$ ist die Menge der Konfigurationen (σ', a', pc'), wobei $\sigma' \in \Sigma$, $pc' \in N$ und a' eine tFUP-Anweisung ist, die anhand der textFUP-Anweisungen folgendermaßen erhalten wird: Jede textFUP-Zuweisung der Form $\_L_i = f_i(x)$ (wobei $\_L_i$ eine Leitungsvariable ist und f_i ein Ausdruck mit einer Reihe von

Argumenten x) erscheint unter den tFUP-Anweisungen nicht. Eine solche Zuweisung beeinflusst die Konstruktion einer tFUP-Anweisung folgendermaßen: Aus einer textFUP-Anweisung a, die keine Zuweisung von Leitungsvariablen ist, wird eine tFUP-Anweisung folgendermaßen konstruiert:

  - Falls a keine Leitungsvariable als Argument hat, wird a gleichzeitig eine tFUP-Anweisung.
  - Andererseits, falls a eine Leitungsvariable $\_L_i$ enthält ($\_L_i = f_i(x)$), dann ist $a(f_i(x))$ eine tFUP-Anweisung, bei der jedes Auftreten von $\_L_i$ durch $f_i(x)$ umgesetzt wird.

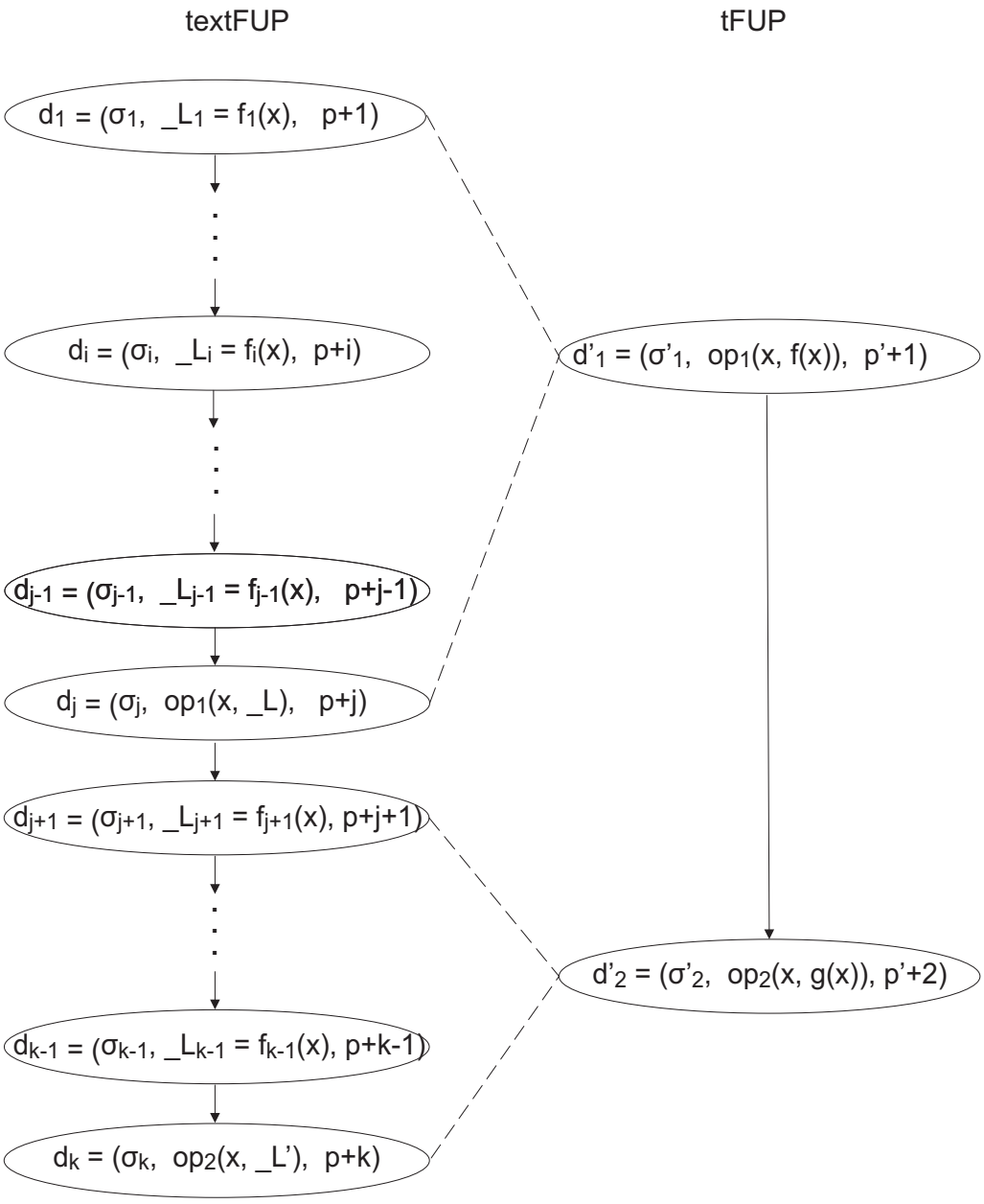
- $d'_0 = (\sigma'_0, a'_0, 1)$ ist die Startkonfiguration.
- Der Programmzähler wird sukzessiv inkrementiert und für zwei Konfigurationen $d' = (\sigma', a', p')$ und $g' = (\gamma', b', q')$ aus $\mathcal{S}'$ gilt $d' \hookrightarrow' g'$ genau dann, wenn $\gamma' = a'(\sigma')$ und $q' = p' + 1$.

Es wird nun die Frage betrachtet, ob $\mathcal{S}$ und $\mathcal{S}'$ äquivalent sind. Eine graphische Darstellung des Verhaltens beider Transitionssysteme bezüglich Leitungsvariablen wird in Abbildung 5.3 gezeigt. Da diese Systeme verschiedene Kardinalitäten haben, kann man über eine Isomorphie zwischen den Systemen nicht sprechen. Allerdings könnte eine starke Bisimulation betrachtet werden, deren Definition für die betrachtete Transitionssysteme folgendermaßen formuliert werden kann ([Tau89]).

Definition 5.6. *(Starke Bisimulation)*
Seien N ein FUP-Netzwerk, $\mathcal{S} = (\mathcal{D}, d_0, \hookrightarrow)$ das entsprechende textFUP-Transitionssystem und $\mathcal{S}' = (\mathcal{D}', d'_0, \hookrightarrow')$ das entsprechende tFUP-Transitionssystem. $\mathcal{S}$ und $\mathcal{S}'$ heißen stark bisimular ($\mathcal{S} \sim \mathcal{S}'$), wenn es eine Relation $B \subseteq \mathcal{D} \times \mathcal{D}'$ gibt, die eine starke Bisimulation für (d_0, d'_0) ist. Das heißt $(d_0, d'_0) \in B$ und für alle $(d, d') \in B$ gilt

- $d \hookrightarrow g \in \mathcal{D} \Rightarrow \exists g' \in \mathcal{D}'$ *mit* $d' \hookrightarrow' g'$ *und* $(g, g') \in B$
- $d' \hookrightarrow' g' \in \mathcal{D}' \Rightarrow \exists g \in \mathcal{D}$ *mit* $d \hookrightarrow g$ *und* $(g, g') \in B$.

Anschaulich ist eine Bisimulation für zwei Transitionssysteme eine paarweise Zuordnung von Zuständen der beiden Systeme, die man als „gleich“ identifiziert. Im Beispiel von textFUP- und tFUP-Transitionssystem kann man diese „Gleichheit“



$x = (x_1, ..., x_n)$, x_i - Variable; $\_L = (\_L_1, ..., \_L_{j-1})$; $\_L' = (\_L_{j+1}, ..., \_L_{k-1})$;
$f(x) = (f_1, ..., f_{j-1})(x)$; $g(x) = (f_{j+1}, ..., f_{k-1})(x)$

Abbildung 5.3: Substitution von Leitungsvariablen

zwischen mehreren textFUP- und einem tFUP-Zustand betrachten. Eine solche Betrachtung wird in Abbildung 5.3 graphisch dargestellt, in dem Systemzustände mit Leitungsvariablen ($d_1,\ d_i,\ d_{j-1},\ d_{j+1},\ d_{k-1}$) und ohne Leitungsvariablen ($d_j,\ d_k$) gegenüber gestellt werden.

Diskussion über Bisimularität

Sei d_j die Konfiguration mit der ersten textFUP-Anweisung, die keine Zuweisung einer Leitungsvariable ist und x und $\_L_1$ als Operanden hat ($a = op(x, \_L_1)$). Im Allgemeinen kann diese Konfiguration mehr als eine Variable ($x = (x_1, \ldots, x_n)$) und mehr als eine Leitungsvariable ($\_L = (\_L_1, \ldots, \_L_m)$) umfassen (siehe Abbildung 5.3). Hier wird nur eine Variable betrachtet, um die Erklärung einfacher zu machen. Dann kann man die entsprechende tFUP-Anweisung als $a' = op_1(x, f_1(x))$ darstellen, wobei $\_L_1 = f_1(x)$. Sei $\sigma_1 = \sigma'_1$. Wenn $\sigma_{j+1} = a[\sigma_j]$ und $\sigma'_2 = a'[\sigma'_1]$, gilt im Allgemeinen $\sigma_{j+1} = \sigma'_2$ nicht. Das kann man in Abhängigkeit davon betrachten, ob x eine Eingangs-, Ausgangs- oder eine lokale Variable des Operators op_1 bzw. f_1 ist. Ein Ausgang eines Operators, kann nur eine Leitungsvariable sein. Sogar wenn der Ausgang einer Variable zugewiesen werden soll, wird dafür ein Zuweisungsoperator verwendet, der die Variable als eine lokale Variable hat. Deswegen wird im Weiteren x nicht als Ausgangsvariable betrachtet. Dann bleiben folgende Möglichkeiten:

- x ist eine Eingangsvariable von op_1 und
  - x ist eine Eingangsvariable von f_1
    Es gilt $\sigma_{j+1} = a[\sigma_j] = op_1(x, \_L_1)[\sigma_j]$. Da x eine Eingangsvariable von f_1 ist, ist der Wert von x in σ_j, σ_1 und σ'_1 gleich. Dann gilt: $\_L_1[\sigma_j] = f_1(x)[\sigma_1] = f_1(x)[\sigma'_1]$, d.h. $\sigma_j = \sigma'_2$.
  - x ist eine lokale Variable von f_1
    Das heißt, dass der Wert von x durch den Operator f_1 geändert werden kann. Dann wird x in σ_j und σ'_1 nicht gleich belegt. Trotz $\_L_1[\sigma_j] = f_1(x)[\sigma_1] = f_1(x)[\sigma'_1]$ gilt dann $\sigma_{j+1} = a[\sigma_j] \neq a'[\sigma'_1] = \sigma'_2$.
- x ist eine lokale Variable von op_1 und
  - x ist eine Eingangsvariable von f_1
    In diesem Fall gilt $\sigma_{j+1} = \sigma'_2$. Dies wird zum Problem, wenn $\_L_1$ inner-

halb eines weiteren Operators $op_2(x, \_L_1)$ verwendet wird. Dann hat x nämlich für σ_k und σ'_2 den gleichen Wert. Jedoch gilt dies für $\_L_1$ und $f_1(x)$ nicht: $\_L_1[\sigma_k] = f_1(x)[\sigma_i] = f_1(x)[\sigma'_1] \neq f_1(x)[\sigma'_2]$.

– x ist eine lokale Variable von f_1
  Ähnlich zum vorherigen Fall gilt auch hier $\sigma_k \neq \sigma'_2$.

Aus der vorherigen Diskussion folgt, dass eine Anweisung zum Problem wird, wenn in dieser sowohl eine Leitungsvariable als auch ein Operand mit lokalen Variablen verwendet wird. Diese Betrachtung kann durch die folgenden Beispiele veranschaulicht werden.

Beispiel 5.1. *(Leitungsvariable am Eingang eines Operators mit lokalen Variablen) Seien x und y zwei booleschen Variablen, die konjugiert werden. Wenn das Ergebnis true ist, wird x durch den R-Operator auf false gesetzt. Das gleiche wird mit y gemacht. Mit einer Belegung von x und y mit true erhält man bei tFUP nicht das erwartete Ergebnis (siehe Abbildung 5.4).*

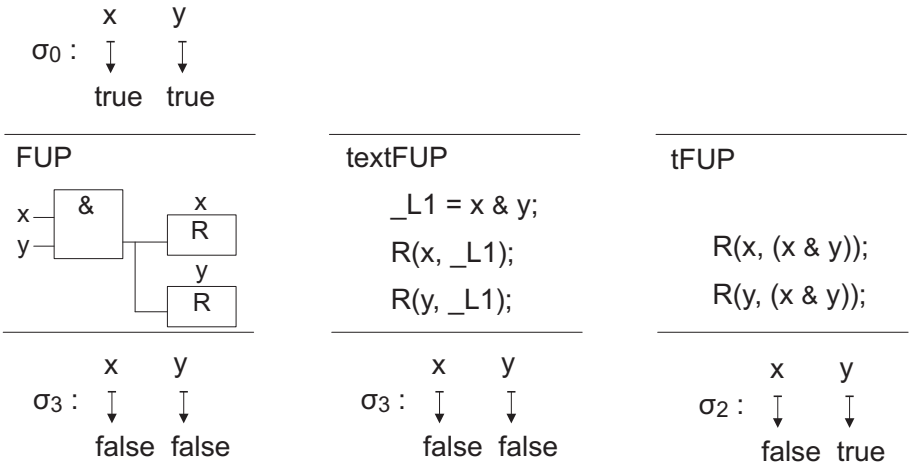


Abbildung 5.4: Beispiel 1: FUP, textFUP und tFUP im Vergleich

Die in Abbildung 5.5 dargestellte Verknüpfung von FUP-Elementen wird in der Praxis vermieden. Allerdings wird in folgendem Beispiel eine zulässige Kombination von FUP-Elementen dargestellt, die die vorherige Betrachtung übersichtlicher macht.

Beispiel 5.2. *(Leitungsvariable am Ausgang eines Operators mit lokalen Variablen) Seien x, y, a und b vier booleschen Variablen, die mit true, false, false und false belegt sind. Wenn ein SR-Operator (Flipflop setzen/rücksetzen) mit x und y als*

Eingang verwendet wird (siehe Abbildung 5.5), werden sowohl a als auch die Leitungsvariable $\_L_1$ auf true gesetzt. Damit bekommen auch $\_L_2$ und b den Wert true. Da bei tFUP im Ausdruck $a \;\&\; SR(a, x, y)$ jeweils der Anfangswert von a (false) benutzt wird, wurde b bei tFUP auf false gesetzt.

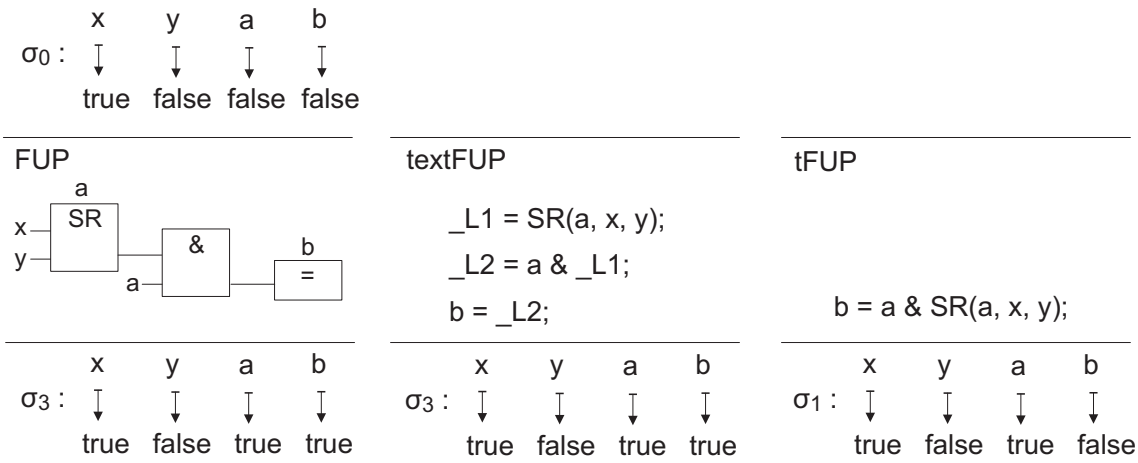


Abbildung 5.5: Beispiel 2: FUP, textFUP und tFUP im Vergleich

Eingeschränkte Substitution

Die Lösung für das Problem bei der Substitution von Leitungsvariablen findet man in einer Einschränkung bei der Verwendung der Substitution. Generell werden Leitungsvariablen benutzt, um die Verbindung zwischen Operatoren zu realisieren. Bei der Substitution von Leitungsvariablen werden Operatoren mit lokalen Variablen zum Problem. Aus Sicht dieser Operatoren gibt es zwei Fälle, in denen eine Leitungsvariable nicht substituiert werden darf:

1. Wenn am Eingang des Operators mit lokalen Variablen die Leitungsvariable benutzt wird. Als Beispiel für diese Situation kann man das in Abbildung 5.4 dargestellte Beispiel anschauen. In dem Beispiel wird die Leitungsvariable $\_L1$ als Operand beim R-Operator benutzt.

2. Wenn ein Ausgang eines Operators mit lokalen Variablen der Leitungsvariable zugewiesen werden soll.

Wenn diese Einschränkungen berücksichtigt werden, kann man die starke Bisimularität zwischen einem textFUP- und einem tFUP-Transitionssystem in folgendem Lemma zeigen.

Lemma 5.2. *Seien N ein FUP-Netzwerk, $\mathcal{S} = (\mathcal{D}, d_0, \hookrightarrow)$ das entsprechende textFUP-Transitionssystem und $\mathcal{S}' = (\mathcal{D}', d_0', \hookrightarrow')$ das entsprechende tFUP-Transitionssystem, wobei $d_0 = (\sigma_0, a_0, 1)$ und $d_0' = (\sigma_0, a_0', 1)$. Sei $B \subseteq \mathcal{D} \times \mathcal{D}'$ eine Relation, die folgendermaßen definiert ist:*

$$\forall d = (\sigma, a, p) \in \mathcal{D} \text{ und } \forall d' = (\sigma', a', p') \in \mathcal{D}' \quad (d, d') \in B$$

genau dann, wenn Folgendes erfüllt ist:

1. *Wenn a eine von den folgenden drei Anweisungen ist:*

   (a) *a ist keine Zuweisung von Leitungsvariablen,*

   (b) *a ist eine Zuweisung von Leitungsvariablen, bei der ein Operator mit lokalen Variablen benutzt wird, oder*

   (c) *a ist eine Zuweisung von einer Leitungsvariable, die bei einem Operator mit lokalen Variablen benutzt wird,*

   dann gilt $a' = a$ und $\sigma' = \sigma$.

2. *Andernfalls ist a eine Zuweisung von einer Leitungsvariable mit folgenden Eigenschaften*

   (a) *bei a wird kein Operator mit lokalen Variablen verwendet, und*

   (b) *die Leitungsvariable wird bei keinem Operator mit lokalen Variablen verwendet.*

   Sei $g = (\gamma, b, q)$ die Konfiguration mit dem kleinsten q, die die Bedingung aus (1.) erfüllt. Dann ist die gesuchte Konfiguration d', die Konfiguration aus $\mathcal{D}'$, für die $(g, d') \in B$ gilt.

Dann ist B eine starke Bisimulation.

Beweis. Seien $d = (\sigma, a, p)$ und $d' = (\sigma', a', p')$ zwei Konfigurationen mit der Eigenschaft $(d, d') \in B$. Dann gilt Folgendes:

- Sei $g = (\gamma, b, q) \in \mathcal{D}$ eine Konfiguration, die $d \hookrightarrow g$ erfüllt. Betrachtet wird das Element g' aus $\mathcal{D}'$, für das $d' \hookrightarrow' g'$ gilt. Es sollte gezeigt werden, dass $(g, g') \in B$

gilt. Nach der Definition von B kann für $d = (\sigma, a, p)$ eine von zwei Situationen auftreten:

- Die Anweisung a ist eine von den drei Anweisungen aus (1.) in der Definition von B. In diesem Fall wird die Anweisung b aus der Konfiguration g betrachtet. Falls b eine Zuweisung einer Leitungsvariable ist, die keinen Verknüpfungspunkt mit einem Operator mit lokalen Variablen hat, erfüllen g und g' die zweite Bedingung aus der Definition von B. Andernfalls, wenn b keine Zuweisung ist, muss noch überprüft werden, ob $\gamma = \gamma'$ erfüllt ist. Dies aber gilt nach dem vorherigen Absatz und der Diskussion über Bisimularität von textFUP und tFUP.

- Die Anweisung a ist eine Zuweisung von einer Leitungsvariable, die keine Verknüpfungspunkte mit einem Operator mit lokalen Variablen hat. Nach der Definition von $\mathcal{S}'$ wird diese Variable in der Konfiguration g' substituiert. Dann erfüllen d' und g' die zweite Bedingung aus der Definition von B.

- Sei g' eine Konfiguration, die $d' \hookrightarrow' g'$ erfüllt. Dann wird dasjenige $g = (\gamma, b, q) \in \mathcal{D}$ betrachtet, welches die Eigenschaft $d \hookrightarrow g$ erfüllt. Ähnlich zur vorherigen Situation können auch hier zwei Möglichkeiten für die Anweisung a auftreten und es kann festgestellt werden, das $(g, g') \in B$ gilt.

□

Beispiel 5.3. *(Starke Bisimulation)*
Die starke Bisimulation für das in Abbildung 5.3 gezeigte Beispiel sieht folgendermaßen aus:

$$B = \{(d_1, d'_1), \ldots, (d_i, d'_1), \ldots, (d_{j-1}, d'_1), (d_j, d'_1), (d_{j+1}, d'_2), \ldots, (d_{k-1}, d'_2), (d_k, d'_2)\}.$$

5.4 Modellierung von FUP-Programmen in NuSMV

Der Model Checker NuSMV wurde von IRST (Instituto per la Ricerca Scientifica e Tecnologica) und CMU (Carnegie Mellon University) entwickelt ([CCGR00]). Dabei geht es um eine Re-Implementierung und Erweiterung von SMV, dem ersten auf BDD basierenden Model Checker.

Komplexe Systeme werden in Module zerlegt, die immer neu instanziert werden können. Damit ist eine modulare Beschreibung von Systemen und Wiederverwendbarkeit von Komponenten möglich (siehe [CCJ+] und [CCK+]). Die wichtigsten Eigenschaften von NuSMV werden im Folgenden anhand von in FUP repräsentierten Modellen dargestellt. Dafür wird das in Abschnitt 4.2.4 eingeführte Beispiel benutzt.

Das Hauptprogramm wird im *MODULE main* zusammengefasst (siehe Abbildung 5.6). Das Modul kann mehrere Abschnitte haben. Für die FUP-Modellierung werden die Abschnitte *VAR*, *DEFINE*, *ASSIGN* und *SPEC* benutzt. Der *VAR*-Abschnitt wird für die Variablendeklaration benutzt. Im *DEFINE*-Abschnitt werden Symbole definiert, die häufig verwendete Ausdrücke im Modell ersetzen können. Im *ASSIGN*-Abschnitt werden Zuweisungen beschrieben. Die *CTL*-Spezifikation des Modells wird im *SPEC*-Abschnitt angegeben. Zunächst wird der Aufbau des *main*-Moduls anhand des in Abbildung 5.6 dargestellten Beispiels beschrieben.

Variablen

Eine Programmvariable, deren Wert während der Programmausführung unverändert bleibt, wie beispielsweise die Anzahl der Programmzeilen im Beispiel MAX_pc, wird im *DEFINE*-Abschnitt eingeführt. Zusätzlich werden in diesem Abschnitt definiert:

- Labels, die Programmzeilen bezeichnen, zu denen im Programm mit Sprunganweisungen gesprungen wird; und
- bestimmte Konstanten, die für die mathematischen Operationen wie z.B. Addition benutzt werden (eine genaue Beschreibung wird unten angegeben).

Die restlichen Programmvariablen werden im *VAR*-Abschnitt deklariert. Für eine boolesche Variable wird einfach ihr Typ angegeben. Im Unterschied dazu muss für eine integer Variable ein Bereich angegeben werden, dem die Variable gehört. Neben Programmvariablen werden im *VAR*-Abschnitt auch der Programmzähler *pc* und

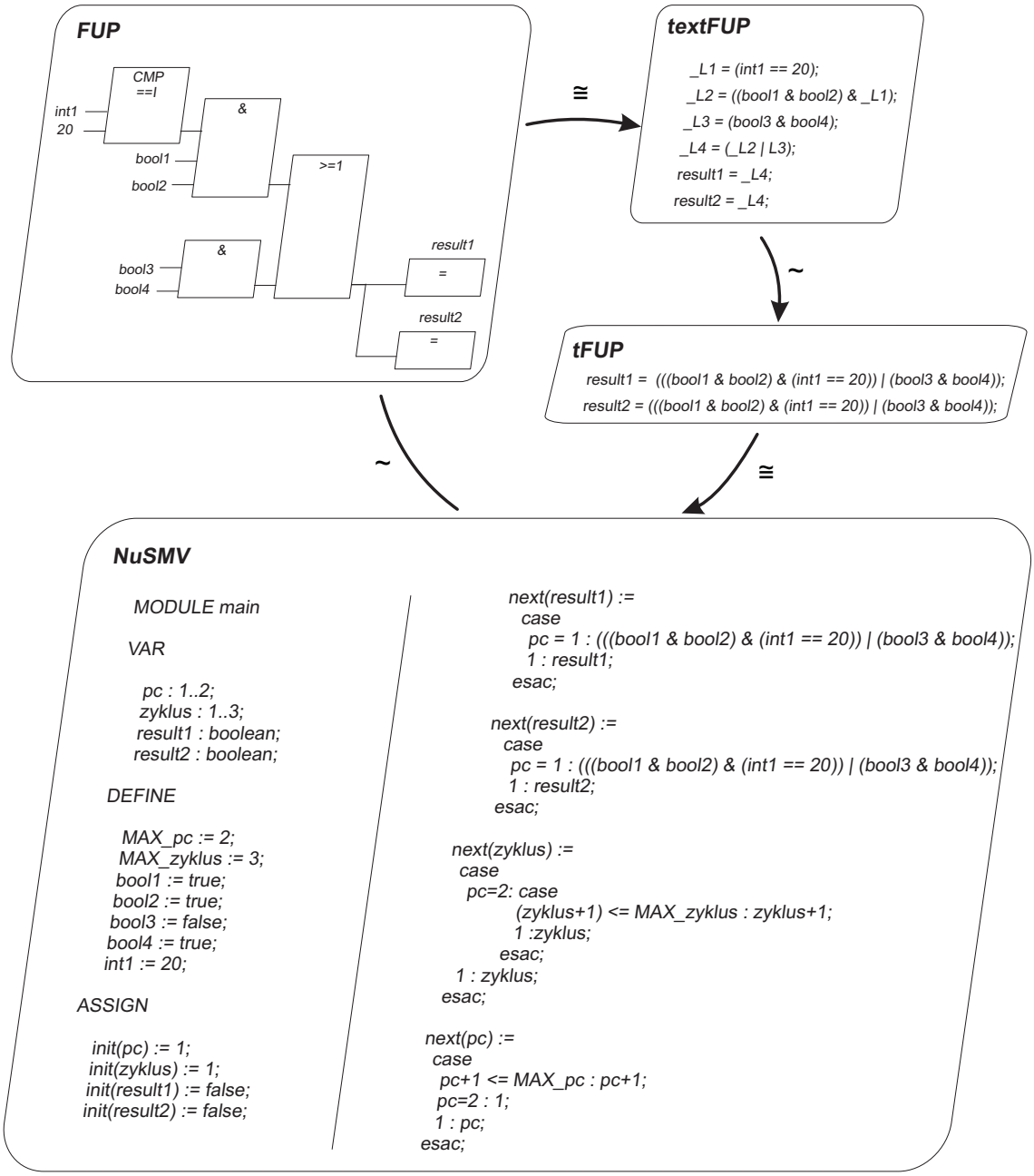


Abbildung 5.6: NuSMV Model

der Zykluszähler *zyklus* beschrieben. Mit dem Programmzähler hält man fest, welche Programmzeile bearbeitet wird. Der Zykluszähler wird bei der Darstellung der Spezifikation durch logische Formeln benutzt, wobei eine Reaktion des Programms nach einer bestimmten Anzahl von Programmzyklen überprüft werden soll.

Transitionen

Die Transitionen des textFUP-Transitionssystems $\mathcal{D}$ werden im *ASSIGN*-Abschnitt des NuSMV-Modells beschrieben. Dafür werden die Konstrukte *init* und *next* verwendet. *init*-Konstrukte werden für die Beschreibung des Startzustands σ_0 von $\mathcal{D}$ benutzt, in dem der Anfangswert jeder Variable definiert wird. *next*-Konstrukte beschreiben das Verhalten von Variablen beim Programmablauf. Der Ablauf des Programms wird durch den Programmzähler beschrieben. Wenn eine Anweisung den Wert einer Variable ändert, wird diese Änderung durch Angabe der Programmzeile, in der sich die Anweisung befindet, und Art der Veränderung angegeben.

Abhängig von der Art der Anweisungen sind bei einigen Übergängen neben dem Programmzähler noch zusätzliche Bedingungen relevant. Ein Beispiel dafür ist die Addition von zwei Integern. Bevor die Summe einer Variable zugewiesen wird, muss zuerst sichergestellt werden, dass die Summe im Wertebereich der Variablen liegt. Beispielsweise muss bei der Bearbeitung jeder neuen Anweisung der Programmzähler um 1 inkrementiert werden. Deswegen muss zuerst überprüft werden, ob $pc + 1 \leq MAX\_pc$ gilt. Erst im positiven Fall darf dem Programmzähler $pc + 1$ zugewiesen werden.

Spezifikation

Bei der Beschreibung von SPS in Kapitel 2 wurde bereits besprochen, dass bei der Ausführung eines SPS-Programms drei Phasen wichtig sind: Lesen von Inputs, Programmausführung, Schreiben von Outputs. Dies wird zyklisch ausgeführt. Die meisten Fragen über Programmeigenschaften sind der Form:

„Welche Werte haben bestimmte Programmvariablen nach der Ausführung einer bestimmten Anzahl von Zyklen?“.

Die Programmeigenschaften dieser Art können analysiert werden, in dem die zwei bereits eingeführten Variablen benutzt werden: 1) Der Programmzähler *pc*, der am Ende einer Ausführung des Programms den Wert MAX_pc erreicht und danach wieder auf 1 gesetzt wird; 2) der Zykluszähler *zyklus*, der am Ende eines Zyklus'

inkrementiert wird. Um die Werte der Variablen nach der Ausführung der Anweisung in der letzten Programmzeile abzufragen, wird MAX_pc auf die Anzahl an Codezeilen plus 1 gesetzt. Nach dieser „imaginären“ Anweisung, mit der die Programmergebnisse angefragt werden können, fängt ein neuer Zyklus an.

Eine mögliche Programmeigenschaft könnte folgendermaßen formuliert werden:

„Am Ende des ersten Zyklus sollten die Variablen $x_1, \ldots, x_n$ *die Werte* $a_1, \ldots, a_n$ *haben“.*

Eine Spezifikation für ein NuSMV-Modell kann in CTL, LTL oder PSL (Property Specification Language) geschrieben werden. Um beispielsweise im Modell eine CTL-Formel aufzuschreiben, muss man vor der Formel das Schlüsselwort *CTLSPEC* angeben. Die gegebene Eigenschaft wird dann in CTL folgendermaßen formuliert:

$$CTLSPEC \quad AG((zyklus = 1 \;\&\; pc = MAX\_pc) \quad -> \\ ((x1 = a1) \;\&\; \ldots \;\&\; (xn = an))).$$

Genauere Informationen hierüber finden sich im folgenden Kapitel, in dem ein umfangreiches Beispiel erläutert wird.

Wiederverwendung von Modulen

Im Allgemeinen kann ein Modul Parameter haben, was eine mehrfache Instanzierung von Modulen ermöglicht. Diese modulare Beschreibung von Komponenten ermöglicht, dass im Programm aufgerufene Funktionen und Funktionsbausteine ebenfalls durch Module repräsentiert werden können. Neben dieser modularen Repräsentation gibt es auch eine sogenannte *flache Repräsentation* von Funktionen. Diese Repräsentationen und ihre Vor- und Nachteile werden zunächst kurz erläutert.

Modulare Repräsentation von Funktionen: Wie bereits angedeutet, kann eine Funktion f mit den Parametern $x_1, \ldots, x_n$ durch ein neues Modul folgendermaßen deklariert werden

$$MODULE \quad f(x1, \ldots, xn)$$

Für einen Aufruf dieser Funktion wird eine Instanz $f1$ der Funktion f im VAR-Abschnitt des *main*-Moduls angelegt mit

$$f1: \quad f(a1, \ldots, an);$$

Ein Vorteil dieser modularen Repräsentation ist, dass sich Funktionen und Funktionsbausteine einfach mit der beschriebenen Technik modellieren lassen. Nachteilig ist jedoch, dass bei jeder neuen Instanzierung eines Moduls der Zustandsraum um die Anzahl der Modulzustände vervielfacht wird.

Flache Repräsentation von Funktionen: Eine andere Möglichkeit zur Modellierung einer Funktion ist, diese Funktion aus der Sicht ihrer Auswirkung auf ihre Outputs zu betrachten. Dazu wird eine Funktion f, die beispielsweise eine Variable x beeinflusst, durch einen Ausdruck $expr(x1,\ \ldots,\ xn)$ dargestellt. Bei einem Aufruf dieser Funktion in Programmzeile l mit Parametern $a1,\ \ldots,\ an$, erscheint Folgendes im $ASSIGN$-Abschnitt bei der Beschreibung von Variable x

$$
\begin{aligned}
next(x)\ := case\\
&\ldots\\
&pc = l\colon\ expr(a1,\ \ldots,\ an);\\
&\ldots\\
esac;
\end{aligned}
$$

Im Gegensatz zur vorherigen Repräsentation steigt bei dieser Technik beim Funktionsaufruf der Zustandsraum nicht auf das Kreuzprodukt von Hauptmodul und Funktionsmodul. Der Nachteil liegt allerdings in der manuellen Modellierung von Funktionen. Von daher ist diese Technik nur für einfache Funktionen geeignet. Wenn die Funktionen im Programm häufig aufgerufen werden, ist diese Technik sehr effizient.

5.5 Zusammenfassung

Nach einer ausführlichen Beschreibung des Verfahrens für die Erstellung eines NuSMV-Modells aus einem FUP-Programm, werden im Folgenden die wichtigsten Schritte zusammengefasst. Eine graphische Repräsentation des Verfahrens wurde in Abbildung 5.6 dargestellt. Kurz gefasst gibt es drei wichtige Punkte:

1. textFUP-Format: Das textFUP-Format eines FUP-Programms ist eine textuelle Darstellung der graphischen SPS-Programmiersprache. In Lemma 5.1 wird gezeigt, dass es eine Isomorphie zwischen dem FUP- und dem textFUP-Transitionssystem gibt.

2. tFUP: Um eine große Zahl an Leitungsvariablen im Zustandsraum des gesuchten NuSMV-Modell zu vermeiden, wird eine Substitution von textFUP vorgeschlagen. Durch Lemma 5.2 wird gezeigt, dass es eine starke Bisimulation zwischen dem textFUP- und dem tFUP-Transitionssystemen gibt.

3. NuSMV: Das NuSMV-Modell ist einfach eine Darstellung des tFUP-Transitionssystems in der Eingabesprache des Model Checkers NuSMV. Damit können tFUP und NuSMV als isomorph betrachtet werden.

In der beschriebenen Transformationskette werden in zwei Schritten isomorphe und in einem Schritt stark bisimulare Systeme erzeugt. Eine Isomorphie ist gleichzeitig eine starke Bisimulation. Da starke Bisimulation eine transitive Relation ist, sind dann das mit dem vorgestellten Verfahren erzeugte NuSMV-Modell und das entsprechende FUP-Programm äquivalent. Genauer ist die Äquivalenz zwischen den zwei Systemen durch eine starke Bisimulation bestimmt.

6 Das Verifikationsverfahren in der Praxis

Nachdem die theoretischen Grundlagen der Methode für die FUP-Verifikation im vorherigen Kapitel vorgestellt wurden, wird in diesem Kapitel ihre Anwendung in der Praxis präsentiert. Es wird eine Fallstudie vorgestellt, mit der die Anwendung der Methode im Bereich der Eisenbahnautomatisierung beschrieben wird (siehe Abbildung 6.1). Dabei wird die FUP-Software betrachtet, die bei einer Stellwerkfamilie benutzt wird. Eine genaue Beschreibung der Funktionalität eines Stellwerks und die Fallstudie folgen im ersten Abschnitt dieses Kapitels.

Allgemein betrachtet, ist zur Zeit die formale Verifikation in der Industrie nicht weit verbreitet. Ein Grund dafür liegt in den komplexen theoretischen Kenntnissen, die hinter der formalen Verifikation stehen und die bei Anwendern sehr oft nicht vorhanden sind. Wenn man ein Verifikationsverfahren in der Praxis anwenden möchte, soll es in dem Maße automatisiert sein, dass nur die für den Anwender wichtigen Aspekte sichtbar sind. Eine Beschreibung der Automatisierung der vorgeschlagene Verifikationsmethode wird im zweiten Abschnitt dieses Kapitels vorgestellt.

6.1 Fallstudie

Stellwerke sind Bahnanlagen, die zur zentralen Stellung von Außenelementen wie Weichen und Signalen dienen. Eine Einführung in die Thematik findet man unter anderem in [Pac08]. Es gibt verschiedene Arten von Stellwerken. Je nach dem mit welcher Technik die Einrichtungen auf den Fahrweg beeinflusst werden, unterscheidet man:

- mechanisches Stellwerk,

Abbildung 6.1: Beispiel eines Bedienraums und eines Stellwerks

- elektromechanisches Stellwerk,
- Relaisstellwerk und
- elektronisches Stellwerk.

Die neuste Stellwerksbauform ist das elektronische Stellwerk (ESTW). Ein Stellwerk dieser Art arbeitet rechnergesteuert. Grundsätzlich besteht ein Stellwerk aus einer Innenanlage und aus einer Außenanlage mit Komponenten wie Weichen, Signale, usw. Bei ESTW wird die Innenanlage in Hardware und Software unterteilt. In der vorliegenden Arbeit wird ausschließlich die Stellwerksoftware betrachtet.

Der Rechnertechnik bei Stellwerken vertraute man lange Zeit nicht. Bei ESTW bestimmt die Software erforderliche Abhängigkeiten zwischen Außenelementen und Stellwerkslogik, die einem Zug freie Fahrt gewährleisten. Damit haben Stellwerke viel an Komplexität gewonnen. Im Gegensatz zu den mechanischen oder Relaisstellwerken war es nun nicht mehr so einfach den Nachweis einer sicheren Funktionsweise der Stellwerke zu führen.

Ein formaler Nachweis über die Sicherheit der Stellwerksoftware kann man mit Hilfe der formalen Verifikation gewinnen. Für die Anwendung von formaler Verifikation in diesem Bereich sprechen zwei wichtige Gründe:

- Sicherheitsaspekte - Technische Fehler bei diesen Anlagen können zu schweren Unfällen führen. Von daher ist sehr wichtig eine fehlerfreie Funktionsweise der Stellwerken abzusichern.

- Effizienz - Die Komplexität der Stellwerksoftware hat schon längst eine Dimension erreicht, bei der es schwierig ist, allein durch das Testen eine fehlerfreie Funktionalität der Software zu gewährleisten.

In der Eisenbahntechnik wird das Konzept der Sicherheitsanforderungsstufe verwendet, die als SIL (*safety integrity level*) bezeichnet wird (siehe dazu [Bra05]). Im Allgemeinen beruht die Sicherheit sowohl auf Maßnahmen zur Vermeidung systematischer Fehler als auch auf Maßnahmen zur Beherrschung zufälliger Ausfälle. Einer Funktion eines Systems wird SIL als Maß für ihre Sicherheit gegen systematischen Fehler zugeordnet. Gemäß der weltweiten Sicherheitsgrundnorm IEC 61508 spricht man über SIL1, SIL2, SIL3 und SIL4 Sicherheitsanforderungsstufe ([Int00]). Zusätzlich kann die Rede von einem Niveau unter SIL1 oder einem Niveau über SIL4 sein. Je höher die Stufe ist, desto höher ist die Sicherheit der entsprechenden Funktion.

Stellwerke sind sicherheitsrelevante Systeme, deren Funktionen unterschiedlichen SILs genügen. Für den Bereich, wo der SIL3 erfüllt werden muss, bietet Firma Siemens das SICAS S7 Stellwerk an. Dieses Stellwerk basiert auf der speicherprogrammierbaren Steuerung SIMATIC S7. Aus diesem Umfeld stammt die folgende Fallstudie. Es wird die Stellwerksoftware betrachtet, die in der graphischen SPS-Programmiersprache FUP geschrieben ist. Mit dem vorgestellten Verfahren für die FUP-Verifikation kann die Korrektheit der Software bewiesen werden und damit könnte ein höherer SIL einer Systemfunktion nachgewiesen werden.

6.1.1 Stellwerksoftware

Die Stellwerksoftware ist in ca. 10 Komponenten aufgeteilt, wobei jede Komponente für die Bearbeitung einzelner Stellwerkfunktionen zuständig ist. Einige Komponenten sind:

- Die Komponente Organisation bildet das Gerüst, in das alle anderen Komponenten eingebunden sind.

- Die Komponente Freimeldung ermittelt, ob ein Bereich besetzt ist und übergibt diese Information an weitere Komponenten.

Abbildung 6.2: Beispiel einer Weiche

- Die Komponente Weiche ist in mehrere Subkomponenten untergliedert. Eine von den Komponenten ist beispielsweise Weichensteuerung. Über diese Komponente und dazugehörige Weichenstellteile werden die Weichen gesteuert.
- Die Komponente Signal übernimmt die Aufgabe der Signalisierung.

Im Folgenden wird die Komponente Weiche näher betrachtet (siehe Abbildung 6.2).

6.1.2 Komponente Weiche der Stellwerksoftware

Wie jede andere Komponente, besteht die Komponente Weiche aus mehreren Codemodulen. Diese Codemodule sind anlagenabhängig und werden für jeweils eine Weiche erstellt. Ein Funktionsbaustein (siehe die Aufbau eines FUP-Programms in Kapitel 2) in so einem Codemodul der Komponente Weiche ist für die Weichensteuerung zuständig. Dieser Funktionsbaustein dient als Fallbeispiel für die Demonstration des vorgeschlagenen Verifikationsverfahrens.

Das Modul für die Weichensteuerung hat mehrere Schnittstellen zu anderen Modulen, die durch Eingangs- und Ausgangsparameter im Modul realisiert werden. Es gibt beispielsweise ein Kommando, dass die Umstellung der Weiche durch bestimmte Eingangsparameter an das Modul auslöst. Das Kommando wird dann vom Modul bearbeitet und ein als zulässig erkanntes Umstellkommando startet einen Weichenumlauf in die vorgegebene Lage. Das heißt, durch bestimmte Ausgangsparameter werden die entsprechenden Module der Komponente darüber informiert.

Aus diesem Funktionsbaustein wird dann mit dem vorgestellten Verfahren das entsprechende NuSMV-Modell abgeleitet. Das Modell wird im folgenden Abschnitt präsentiert. Anschließend wird die dazu passende Spezifikation beschrieben.

Beschreibung	Vorbedingungen	Erwartete Reaktion
Aus einer rechten Lage wird mit einem Umstellkommando das Richtungrelais links angesteuert und der Umstellvorgang wird aktiv	$ModulSS = Rechts$, $KomSS = UmstKom$	$ModulSS = Links + Schr1$
Nach mindestens $20\,ms$ wird das Lagerelais angesteuert	$\Rightarrow$, $iZeit = 20$	$ModulSS = Schr2$
Nach mindestens $30\,ms$ wird der Schutz angesteuert	$\Rightarrow$, $iZeit = 30$	$ModulSS = Schr3$
Nach mindestens $40\,ms$ startet der Weichenumlauf	$\Rightarrow$, $iZeit = 40$	$ModulSS = Schr4$

Tabelle 6.1: Beispiel eines Testfalls: Umstellen der Weiche nach links

6.1.3 Testfallbeschreibung der Komponente Weiche

Die Spezifikation der Stellwerksoftware ist in mehreren Tabellen erfasst, in denen Standardtestfälle beschrieben sind (siehe Tabelle 6.1). Für jede Komponente wird eine Menge von Tabellen erstellt, wobei eine Tabelle einem Codemodul der Softwarekomponente entspricht. So eine Tabelle enthält alle Testfälle eines Codemoduls. Die Testfallbeschreibung dient in der Fallstudie als Basis für die Formulierung der Szenarien, die bei der Modulverifikation benutzt werden.

Beispiel eines Testfalls

Als Beispiel lässt sich in Tabelle 6.1 eine vereinfachte Beschreibung eines Testfalls für die Weichensteuerung sehen. Dabei wird die Stellteilansteuerung beim Umstellen der Weiche nach links in vier Schritten überprüft. Jeder Schritt wird durch eine kurze Beschreibung, eine Definition von Vorbedingungen (Startkonfiguration vom Testfall) und eine Definition von erwarteter Reaktion dargestellt.

Für ein besseres Verständnis von Tabelle 6.1 soll der Begriff *Variablenbereich* näher erläutert werden.

ModulSS - Modulschnittstelle						
	Links	*Rechts*	*Schr*1	*Schr*2	*Schr*3	*Schr*4
bWeichenLage	0	1				
bRichtungRechts	0	1				
bRichtungLinks	1	0				
bWiederholung			1	0		
bUmstellAktiv			1			
bWeichenLageRelais				1	0	
bSchutz					1	0
iZeit						0

KomSS - Kommandoschnittstelle	
	UmstKom
bKomAktiv	1
bFunktion	1
iElement	100

Tabelle 6.2: Definition von Schnittstellen - Variablenbereichen

Unterschied zwischen einer Variable und einem Variablenbereich In der Beschreibung von Vorbedingungen und erwarteter Reaktion eines Testfalls werden neben Programmvariablen (wie z.B. *iZeit*) auch Variablenbereiche benutzt (wie z.B. *ModulSS* oder *KomSS*). Ein Variablenbereich umfasst eine Beschreibung mehrerer Programmvariablen. Um diese zwei Begriffe voneinander besser zu unterscheiden, wird hier eine Schreibweise von Programmvariablen eingeführt, indem der Name einer Booleschen Programmvariable mit *b* und einer Integervariable mit *i* anfängt.

Dies kann am in Tabelle 6.2 definierten Beispiel des Variablenbereichs *ModulSS* näher erklärt werden. Diesem Variablenbereich gehören folgende Variablen an: *bWeichenLage*, *bRichtungRechts*, *bRichtungLinks*, *bWiederholung*, *bUmstellAktiv*, *bWeichenLageRelais*, *bSchutz* und *iZeit*. Wenn im Testfall geschrieben wird, dass beispielsweise *ModulSS* den Wert *Rechts* haben soll, soll dann Folgendes gelten: $bWeichenLage = 1, bRichtungRechts = 1$ und $bRichtungLinks = 0$.

Variablenbereiche werden für die Beschreibung von Schnittstellen benutzt. Wie

in Tabelle 6.2 dargestellt, wird die Modulschnittstelle durch den Variablenbereich $ModulSS$ und Variablen $bWeichenLage$, $bRichtungRechts$, $bRichtungLinks$, usw. beschrieben. Der Variablenbereich enthält in verschiedenen Testfällen verschiedene Variablenbelegungen. Für jeden Variablenbereich wird eine Aufzählung von Variablenbelegungen definiert. Es werden die Belegungen ausgewählt, die für die Testfallbeschreibungen benutzt werden. Dies ermöglicht eine klare Beschreibung von Testfällen.

Variablenbelegungen für $ModulSS$ werden beispielsweise folgendermaßen definiert:

$$ModulSS \in \{Links,\ Rechts,\ Schr1,\ Schr2,\ Schr3,\ Schr4,\ \ldots\}$$

So wird die Vorbedingung im ersten Schritt des Testfalls (das Richtungsrelais muss in der rechten Lage sein) mit $ModulSS = Rechts$ darstellt. Damit ist gemeint, dass vor der Ausführung des Testfalls $bWeichenLage = 1$, $bRichtungRechts = 1$ und $bRichtungLinks = 0$ gelten muss (siehe Tabelle 6.2). Ähnlich wird am Anfang des Testfalls $KomSS$ durch $UmstKom$ definiert. Wenn eine Schnittstelle in der Vorbedingung nicht spezifiziert wird, wird sie mit ihrem initialen Wert belegt, der wiederum am Anfang der Testtabelle spezifiziert werden muss.

Das Zeichen „⇒“

Am Anfang der Beschreibung der Vorbedingung vom zweiten, dritten und vierten Schritt des Testfalls findet man das Zeichen „⇒“. Damit ist gemeint, dass sich der Testfall auf den vorherigen Testfall aufbaut. Es heißt beispielsweise im Fall vom zweiten Schritt, dass alle Variablen außer $iZeit$ die Werte aus der Reaktion des ersten Schrittes haben.

Die erwartete Reaktion des ersten Testfallschrittes ist durch $ModulSS = Links + Schr1$ beschrieben. Das heißt: $bWeichenLage = 0$, $bRichtungRechts = 0$, $bRichtungLinks = 1$, $bWiederholung = 1$ und $bUmstellAktiv = 1$.

Bei einer symbolischen Addition von zwei (oder mehreren) Werten eines Variablenbereichs ist es möglich, dass eine Variable in der Beschreibung beider (bzw. aller) Elementen der Summe vorhanden ist. In dem Fall soll die Variable mit dem Wert, der im letzten Element vorgeschrieben ist, belegt werden. Es wird beispielsweise $bWeichenLage$ in $Links + Rechts$ mit 1 vorbelegt.

In der Beschreibung der Vorbedingung eines Testfalls kann neben einer Belegung eines Variablenbereichs auch eine Belegung einzelner Variablen erscheinen. Es wer-

den beispielsweise verschiedene Belegungen der Variable *iZeit* im zweiten, dritten und vierten Schritt des Testfalls definiert. Das gleiche gilt auch für die Beschreibung der erwarteten Reaktion.

SPS-Anlauf

Beim Testen von SPS-Software bezieht sich im Allgemeinen ein kleiner Teil der Testfälle einer Komponente auf einen Neustart der SPS. Bevor die CPU nach dem Einschalten mit der Bearbeitung des Anwenderprogramms beginnt, wird ein Anlaufprogramm durchlaufen. Dieses Programm bildet eine Schnittstelle zwischen dem Betriebssystem und dem Anwenderprogramm. Im Anlaufprogramm können bestimmte Voreinstelleungen für das Anwenderprogramm festgelegt werden ([Sie04c]). Ungefähr 5 % der Eigenschaften bezieht sich auf den SPS-Anlauf. Diese Eigenschaften werden mit der vorgestellten Verifikationsmethode nicht überprüft. Es werden nur Eigenschaften betrachtet, die in einer direkten Verbindung mit dem betrachteten Anwenderprogramm stehen.

6.1.4 NuSMV-Modell der Komponente Weiche

Ein kleiner Ausschnitt aus dem NuSMV-Modell der Komponente Weiche wird in Tabelle 6.3 präsentiert. Es werden nur Zeilen aus dem tFUP-Modell dargestellt, bei denen die Belegung der Programmvariablen *bWeichenLage*, *bRichtungRechts*, *bRichtungLinks*, *bUmstellAktiv*, *bWiederholung*, *bKomAktiv*, *bFunktion*, und *iElement* im Programm geändert wird. Diese sind nämlich Variablen, die bei der Spezifikationsbeschreibung benutzt werden. Variablen, deren Rolle für die folgende Erläuterungen irrelevant sind, werden in Tabelle 6.3 durch var_i bezeichnet.

Sowohl bei der Modellbeschreibung als auch bei der Vorstellung der Spezifikation werden die Daten in einer geänderten Form dargestellt, um den Bezug auf der originalen Software zu verbergen. Die Regeln für die Transformation eines FUP-Programms in die Eingabesprache vom Model Checker NuSMV wurden bereits in Kapitel 5 ausführlich beschrieben. In diesem Abschnitt wird erläutert, wie sich Spezifikationen mit Hilfe von logischen Formeln darstellen lassen. Die Umsetzung der Automatisierung dieser Transformationen wird in Abschnitt 6.2 erläutert.

```
DEFINE
bKomAktiv := 1;
bFunktion := 1;
iElement := 100;

ASSIGN
init(bWeichenLage) := 1;
next(bWeichenLage) :=
  case
    pc = 88 : var1;
    pc = 120 & var2 : 1;
    pc = 131 & var2 : 0;
    1 : bWeichenLage;
  esac;
init(bRichtungRechts) := 1;
next(bRichtungRechts) :=
  case
    pc = 104 : bWeichenLage;
    1 : bRichtungRechts;
  esac;
init(bRichtungLinks) := 0;
next(bRichtungLinks) :=
  case
    pc = 105 : !var2;
    1 : bRichtungLinks;
  esac;
init(bWiederholung) := 0;
next(bWiederholung) :=
  case
    pc = 122 & var3 : 1;
    pc = 123 & ((var4 | !var5) | (((var6 & !var7) & var8) | ((var7 & !var6) & var9))) : 0;
    1 : bWiederholung;
  esac;
next(bUmstellAktiv) :=
  case
    pc = 48 & (var10 | var11 | var12) : 0;
    pc = 50 & ((((var13 & !var14) & (var15 & !var16)) | var17) | var18) : 0;
    pc = 52 & ((var13 & !var14) & (var19 > var20)) : 0;
    pc = 96 : ((var21 & !var22) | ((!var21 & !var23) | var24));
    1 : bUmstellAktiv;
  esac;
```

tFUP

```
...
48  R(bUmstellAktiv, (var10 | var11 | var12));
...
50  R(bUmstellAktiv, ((((var13 & !var14) & (var15 &
    !var16)) | var17) | var18));
...
52  R(bUmstellAktiv, (var13 & !var14) & (var19 > var20));
...
88  bWeichenLage = var1;
...
96  bUmstellAktiv = ((var21 & !var22) | ((!var21 & !var23)
    | var24));
...
104  bRichtungRechts = bWeichenLage;
105  bRichtungLinks = !var2;
...
120  S(bWeichenLage, var2);
...
122  S(bWiederholung, var3);
123  R(bWiederholung, ((var4 & !var5) | ((var6 & !var7)
     & var8) | ((var7 & !var6) & var9)));
...
131  R(bWeichenLage, var2);
...
```

Tabelle 6.3: NuSMV-Modell der Komponente Weiche

6.1.5 Beschreibung der Verifikationsszenarien

In der Fallstudie werden Verifikationsszenarien mittels CTL beschrieben. Eine kurze Einführung in die Thematik wurde in Abschnitt 5.4 („Modellierung von FUP-Programmen in NuSMV“) gegeben. Dabei wurden zwei wesentliche Parameter vorgestellt: 1) der Programmcounter *pc*, der am Ende eines Programmzyklus den Wert MAX_pc erreicht, und danach wieder auf 1 gesetzt wird; und 2) der Zykluscounter *zyklus*. Wie bereits angekündigt, dient die Testfallbeschreibung einer Komponente als Basis für die Formulierung der Verifikationsszenarien. Da ein Testfall durch Vorbedingungen und erwartete Reaktion definiert wird, lässt sich das durch das folgende Szenario repräsentieren:

$$\begin{aligned} &\text{„}Vorbedingungen\text{“} \quad \rightarrow \\ &\quad AG(\ (zyklus = 1\ \&\ pc = MAX\_pc) \quad \rightarrow \quad \text{„}Erwartete\ Reaktion\text{“}) \end{aligned} \tag{6.1}$$

Wenn es nicht anders spezifiziert ist, bezieht sich standardmäßig ein Szenario auf einen Programmzyklus. Analog zur schrittweisen Beschreibung eines Testfalls in Tabelle 6.1, wird ein Szenario durch mehrere CTL-Formeln beschrieben.

$$\begin{aligned} &ModulSS = Rechts\ \&\ KomSS = UmstKom \quad \rightarrow \\ &\quad AG(\ (zyklus = 1\ \&\ pc = MAX\_pc)\ \rightarrow\ ModulSS = Links + Schr1) \end{aligned} \tag{6.2}$$

Nachdem die Beschreibung der Schnittstellen in Tabelle 6.2 dargestellt wird, lässt sich der erste Schritt des Szenarios durch folgende CTL-Formel ausdrücken:

$$\begin{aligned} &(bWeichenLage = 1\ \&\ bRichtungRechts = 1\ \&\ bRichtungLinks = 0\ \& \\ &\ bKomAktiv = 1\ \&\ bFunktion = 1\ \&\ iElement = 100) \quad \rightarrow \\ &\quad AG(\ (zyklus = 1\ \&\ pc = MAX\_pc) \quad \rightarrow \\ &\qquad (bWeichenLage = 0\ \&\ bRichtungRechts = 0\ \&\ bRichtungLinks = 1\ \& \\ &\qquad\ bWiederholung = 1\ \&\ bUmstellAktiv = 1)) \end{aligned} \tag{6.3}$$

Ähnlich werden die restlichen drei Formeln kreiert. Wie vorher erwähnt, wird mit „$\Rightarrow$“ gekennzeichnet, dass die Vorbedingungen des aktuellen Schrittes mit der erwarteten Reaktion des vorherigen Schrittes addiert werden. Am Beispiel des zweiten Schrittes heißt dies:

$$\begin{aligned} &(ModulSS = Links + Schr1\ \&\ iZeit = 20) \quad \rightarrow \\ &\qquad AG(\ (zyklus = 1\ \&\ pc = MAX\_pc) \quad \rightarrow \quad ModulSS = Schr2) \end{aligned}$$

Alternative Darstellung der Spezifikation

Wenn in einem Modell nur eine Formel überprüft werden soll, können die Variablenbelegungen aus der Beschreibung der Vorbedingungen als Initialwerte der Variablen im Modell betrachtet werden. Damit lässt sich ein Szenario durch folgende Formel beschreiben:

$$AG(\ (zyklus = 1\ \&\ pc = MAX\_pc) \quad \rightarrow \quad \text{„}Erwartete\ Reaktion\text{“}) \tag{6.4}$$

Mit dieser Vorgehensweise wird ein etwas anderes NuSMV-Modell gebaut, bei dem die Menge der initialen Zustände im Modell reduziert wird. Als Beispiel wird in Tabelle 6.3 der Teil des Modells dargestellt, der sich auf einige von den Variablen aus dem Beispieltestfall bezieht. Dort sind die Initialwerte der Variablen *bWeichenLage*, *bRichtungRechts* und *bRichtungLinks* durch das *init*-Konstrukt definiert. Im Gegensatz dazu sind die Variablen *bKomAktiv*, *bFunktion* und *iElement* im *DEFINE*-Abschnitt definiert, da ihre Werte im Modell unverändert bleiben.

Vor- und Nachteile dieser beiden Ansätze für den Modellaufbau werden im Folgenden auf Basis der Verifikationsergebnisse diskutiert.

6.1.6 Verifikationsergebnisse

Bevor die Model Checking Ergebnisse präsentiert werden, sollen zunächst die grundlegenden Schritte bei der Verifikation durch NuSMV näher beschrieben werden.

Verifikation durch NuSMV

Eine detaillierte Beschreibung des Model Checker NuSMV findet man in [CCJ+] und [CCK+]. Die wichtigsten Eigenschaften werden von den Autoren in [CCGR00] zusammengefasst. Die Grundschritte der NuSMV-Verifikation können erläutert werden, in dem man bestimmte Kommandos aus dem interaktiven Modus von NuSMV beschreibt. Allgemein gesehen, kann der Verifikationsprozess als eine Sequenz folgender Schritte betrachtet werden:

1. *read_model* - Im ersten Schritt der Verifikation wird das Modell gelesen. Es wird eine interne Repräsentation vom Modell gemacht und gespeichert.

2. *flatten_hierarchy* - Im zweiten Schritt wird die hierarchische Repräsentation des Modells durch eine abgeflachte Repräsentation dargestellt. Dabei wird ein Modell durch nur ein Modul repräsentiert, in dem alle aufgerufene Module und Prozesse instanziert werden.

3. *encode_variables* - Danach werden BDD-Variablen generiert.

4. *build_model* - Mit diesem Kommando wird die abgeflachte Repräsentation vom Modell durch BDD dargestellt.

5. *check_ctlspec* - Nachdem die BDD-Darstellung des Modells generiert wurde, kann die Spezifikation überprüft werden. Die CTL-Spezifikation wird beispielsweise mit diesem Kommando überprüft.

Ergebnisse

Das textFUP-Format der betrachteten Softwarekomponente hat 165 Codezeilen. Es werden ca. 100 Variablen benutzt (ungefähr 90 Boolesche und 10 Integervariablen). Die Modellverifikation wurde auf einem Intel(R) Xeon(R) CPU 5150 Rechner mit $2,66\,GHz$ und $3,25\,GB$ RAM durchgeführt.

Die Ausführungszeiten der oben aufgeführten Verifikationsschritte unterscheiden sich wesentlich. Die ersten drei Schritte sind relativ kurz:

1. Auf ein Zehntel der Sekunde abgerundet wird das Modell in $0,0\,s$ gelesen.

2. Die abgeflachte Repräsentation vom Modell wird in ca. $0,2\,s$ gebaut.

3. Die BDD-Variablen werden in ca. $1,3\,s$ generiert.

Die anderen zwei Schritte, die Erstellung vom BDD-basierten Modell und die Spezifikationsüberprüfung, werden anschließend näher betrachtet.

Im vorherigen Abschnitt wurden zwei Vorgehensweisen für den Aufbau des NuSMV-Modells vorgestellt. Sie können von einander am Besten unterschieden werden, in dem man die Formulierung der Spezifikation im Modell betrachtet. Die Ausführungszeiten der ersten drei Verifikationsschritte unterscheiden sich dabei nicht. Im Gegensatz dazu kann man schon auf einen Unterschied bei der Erstellung vom BDD-basierten Modell und der Spezifikationsüberprüfung hinweisen, was in Folgendem diskutiert wird.

Ein Modell für alle Szenarien

Durch Formeln 6.1, 6.2 und 6.3 wird beschrieben, wie die Spezifikation für die erste Variante vom NuSMV-Modell gebaut wird. Die Vorbedingungen eines Szenarios erscheinen dabei im ersten Teil der entsprechenden Formel. Damit werden die Initialwerte der betroffenen Variablen nicht im Modell definiert. Von ca. 10^{65} Zuständen sind ca. 10^{14} Zustände im Modell erreichbar. Die Erstellung des BDD-basierten Modells dauert in diesem Fall ca. $2200\,s$. Für die Spezifikationüberprüfung werden 30 bis $80\,s$ pro Formel gebraucht.

Ein Modell für ein Szenario

Im zweiten Fall wird ein Modell für ein Szenario generiert. Dabei werden die Vorbedingungen für das Szenario durch Initialisierung vom Modell angegeben. Die Spezifikation wird dann in der Form der Formel 6.4 definiert. Im NuSMV-Modell sind dann ca. 6000 von 10^{60} Zuständen erreichbar. Die Erstellung des BDD-basierten Modells dauert in diesem Fall ca. $2500\,s$. In diesem Fall dauert die Spezifikationsübeprüfung unter $1\,s$.

6.2 Automatisierung des Verifikationsverfahrens

Wie bereits besprochen, wird die CTL-Spezifikation anhand der Testfallbeschreibung und der Beschreibung der Modulschnittstellen erstellt (siehe Abbildung 6.3). CTL-Formeln werden einfach gebaut, in dem die Informationen aus verschiedenen Tabellen gesammelt und richtig kombiniert werden. Im Gegensatz dazu ist die Erstellung des NuSMV-Modells etwas komplizierter.

Mit dem vorgestellten Verfahren wird ein beliebiges FUP-Programm so modelliert, dass es sich mit dem Model Checker NuSMV verifizieren lässt. Wie bereits in Kapitel 5 beschrieben, sollte man vor der Erstellung des NuSMV-Modells zuerst das textFUP- und danach das tFUP-Format des FUP-Programms generieren. Eine genaue Beschreibung dieser Schritte wird im Folgenden gegeben. Die graphische Darstellung des Verifikationsprozesses aus Abbildung 5.1 wird in einer detaillierten Form in Abbildung 6.3 präsentiert.

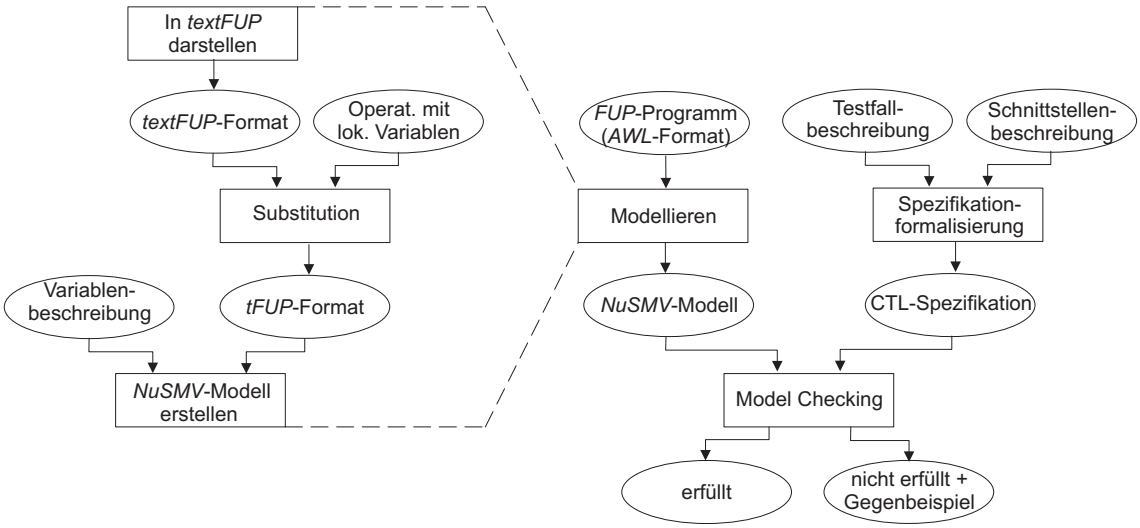


Abbildung 6.3: Erstellen von tFUP

6.2.1 Erstellen des textFUP-Formats

Der erste Schritt bei der Verifikation eines FUP-Programms ist die Erstellung seines textFUP-Formats. Diese textFUP-Darstellung wird aus der AWL-Darstellung des Programms generiert. AWL ist die maschinenorientierte SPS-Programmierungsprache und jedes FUP-Programm kann in AWL abgebildet werden. Die Bandbreite der textFUP-Anweisungen ist der Bandbreite der FUP-Anweisungen äquivalent (siehe [Sie04b]).

Das AWL-Format eines FUP-Programms kann in textFUP mithilfe der kontextfreien Grammatik umgewandelt werden, die in Anhang A vorgestellt ist. Dort werden alle FUP-Anweisungen berücksichtigt, die für die Implementierung der Softwarekomponente Weiche benötigt werden. Um die Vorgehensweise der Umwandlung an dieser Stelle näher zu beschreiben, wird ein kleiner Ausschnitt aus der Grammatik näher erklärt (siehe Tabelle 6.4). Es werden die Grammatikregeln vorgestellt, die man für die Transformation des in Abbildung 6.4 gegebenen Netzwerks braucht.

Das Beispielnetzwerk kann man als eine komplexe Anweisung betrachten, die aus einer bitlogischen Operation besteht, nach der zwei Zuweisungen folgen (siehe Regel (1) in Tabelle 6.4). Eine bitlogische Verknüpfung hat immer einen Output. Da im Netzwerk keine freie Leitungen möglich sind, muss dieser Output irgendwie verbraucht werden. Das heißt, jede bitlogische Verknüpfung endet mit einer Zuweisung, einer Set-Reset Anweisung, einem Sprung oder ihr Output wird in ein Memorybit

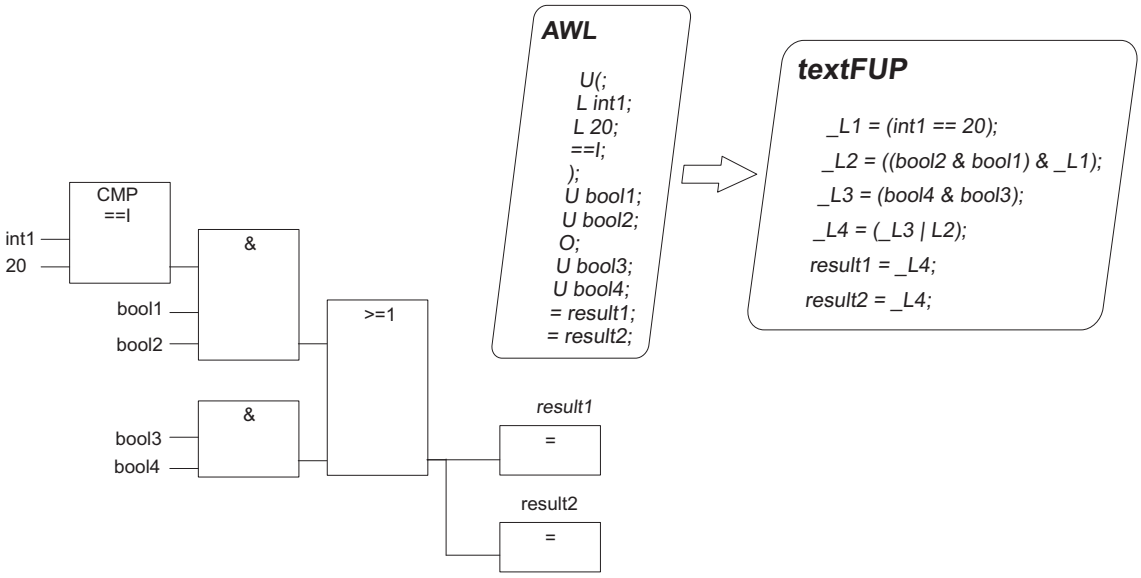


Abbildung 6.4: Beispiel: AWL- und textFUP-Format eines FUP-Programms

geladen (siehe Anhang A). In diesem Beispiel werden Zuweisungen verwendet (Regel (7)). Um die Verwendung von zwei Anweisungen zu ermöglichen, sind Regeln (8) und (9) nötig.

Unter bitlogischen Operationen gehört ein Vergleich von zwei Ganzzahlen (Regel (6)). Wenn man das Ergebnis dieser Operation mit zwei weiteren Booleschen Variablen *UND*-verknüpfen möchte, muss man zuerst Regel (3) und danach zwei mal Regel (4) verwenden. Eine *UND*-Verknüpfung von zwei booleschen Variablen kann man mithilfe von Regeln (2) und (4) erkennen. Regel (5) beschreibt, wie sich zwei bitlogische Operationen mit einer *ODER*-Verknüpfung verbinden lassen.

Für die Erstellung einer textFUP-Datei ist das Konzept der *Leitungsvariablen* sehr wichtig. Diese Variablen werden generiert, wenn Verbindungen zwischen zwei Operanden dargestellt werden sollen. Die Variablen werden als $\_Li$-Variablen gekennzeichnet (siehe Abbildung 6.4). Eine solche Variable wird dann erstellt, wenn die Erkennung eines FUP-Operandes abgeschlossen wird. In diesem Beispiel heißt dies:

- Sobald der Vergleich erkannt wird, wird $\_L1$ erstellt.
- Danach folgt die Erkennung der UND-Verknüpfung (bis die $ODER$-Verknüpfung gelesen wird) mit der anschließenden Erstellung von $\_L2$.
- Um die $ODER$-Verknüpfung auszuführen, braucht man die UND-Verknüpfung

<N-Anweisung>	::= ...	
	| <Bitlogik> <BitlogikEnde>	(1)
<Bitlogik>	::= ...	
	| U <Operand> <Ende>	(2)
	| U(<Ende> <Vergleich>) <Ende>	(3)
	| <Bitlogik> U <Operand> <Ende>	(4)
	| <Bitlogik> O <Ende> <Bitlogik>	(5)
<Vergleich>	::= L <Operand> <Ende> L <Operand> <Ende> COMPARETOK <Ende>	(6)
<BitlogikEnde>	::= ...	
	| <ZuweisungEndeN>	(7)
<ZuweisungEndeN>	::= = <Var> <Ende> <ZuweisungEnde>	(8)
<ZuweisungEnde>	::= | <ZuweisungEnde> = <Var> <Ende>	(9)

Tabelle 6.4: Ein Teil der Grammatik

($\_L3$), die danach folgt.

- Erst dann wird $\_L4$ als Disjunktion von $\_L2$ und $\_L3$ erstellt.

- Am Ende wird das Ergebnis der bitlogischen Operation, das in der Variable $\_L4$ vorliegt, den Variablen $result1$ und $result2$ zugewiesen.

Auf Basis der vorgestellten Grammatik und dem Konzept von Leitungsvariablen werden textFUP-Dateien erstellt.

6.2.2 Erstellen des NuSMV-Modells

Obwohl die erhaltene textFUP-Form des FUP-Programms nach NuSMV transformierbar ist, wird zunächst eine Minimierung des Modells durchgeführt. Wie in Abschnitt 5.3 des vorherigen Kapitel zu finden ist, kann man auf viele Leitungsvariablen im Modell verzichten und damit die Modellgröße reduzieren. Diese Substitution wird im tFUP-Format des FUP-Programms zusammengefasst. Daraus wird das NuSMV-Modell kreiert.

tFUP-Format

Laut der umfangreichen Diskussion über Substitution von Leitungsvariablen (siehe Abschnitt 5.3) braucht man für das Erstellen von tFUP-Format nur die Liste der FUP-Operatoren, die lokale Variablen benutzen. Damit wird bei der Transformation einer textFUP-Datei bei jeder Anweisung überprüft, ob die Anweisung einen Operator aus der Liste verwendet. In einem positiven Fall wird die Anweisung unverändert in die tFUP-Datei geschrieben. Sonst wird zuerst eine Substitution der Leitungsvariable durchgeführt (siehe Abbildung 6.3).

NuSMV-Modell

Die Transformation einer tFUP-Datei in die Eingabesprache von NuSMV wurde bereits in Abschnitt 5.4 beschrieben. Es wurde gezeigt, wie man tFUP-Anweisungen durch NuSMV-Transformationsregeln darstellen kann. Um ein NuSMV-Modell fertigzustellen, braucht man noch die Variablendeklaration (siehe Abbildung 6.3).

Variablendeklaration

In Abschnitt 5.4 wurde bereits die Deklaration von Programmvariablen erwähnt. Um die Abschnitte für die Konstantendefinition (DEFINE) und Variablendeklaration (VAR) komplett zu kreieren, reichen die Variablenbeschreibungen aus der ursprünglichen AWL-Datei eines FUP-Programms nicht. In dem entsprechenden Abschnitt der AWL-Datei findet man den Datentyp einer Variable (bspw. *bool* oder *int*) heraus. Zusätzlich kann bei einer Variablenbeschreibung auch der Initiallwert der Variable stehen. Der VAR-Abschnitt einer AWL-Datei kann beispielsweise folgende Einträge enthalten:

$$\begin{aligned} int1 &: \quad INT; \\ bool1 &: \quad BOOL := FALSE; \end{aligned}$$

Aus dieser Beschreibung kann für den Wertebereich der Integervariable nur der ganze Integerbereich reserviert werden. Damit würde man in einem Programm mit nur wenigen Integervariablen so eine Zustandsraumgröße erreichen, die sehr schnell zu einer Zustandsexplosion bei der Verifikation führen. Aus diesem Grund muss man in einem NuSMV-Modell die Wertebereiche von Variablen so weit wie möglich einschränken. Dies kann nur von einem Fachexperte gemacht werden, der die

Programmeigenschaften gut kennt und die Wertebereiche der Variablen genau definieren kann. Diese Informationen werden in einer Datei zusammengefasst, die neben dem tFUP-Datei für das Erstellen des NuSMV-Modells benötigt wird (siehe Abbildung 6.3).

6.3 Zusammenfassung

In diesem Kapitel wurde die Anwendung des entwickelten Verfahrens auf dem Gebiet von Eisenbahnautomatisierung vorgestellt. In der Fallstudie wurde die Verifikation einer Komponente der Stellwerksoftware betrachtet. Es gibt mehrere wichtige Aspekte für die Anwendung formaler Verifikation:

- Allgemein betrachtet, ist die Automatisierung eines Verifikationsprozesses eine wichtige Voraussetzung für die Anwendung eines Verifikationsverfahrens in der Praxis. Diese Voraussetzung wird beim vorgestellten Verfahren erfüllt, da dabei sowohl die Erstellung eines NuSMV-Modells, als auch die Erstellung der Modellspezifikation automatisiert sind.

- Das Verifikationsverfahren soll in der Lage sein, auf relativ große, praxisrelevante, Modelle anwendbar zu sein. Dieser Aspekt wird mit dem entwickelten Verifikationsverfahren erfüllt, denn die vorgestellte Fallstudie zeigt, dass das Verfahren auf eine Komponente der Stellwerksoftware anwendbar ist.

- Die Dauer der Verifikation soll akzeptabel sein. Die hier vorgestellten Verifikationsergebnisse weisen deutlich höhere Werte bezüglich der Verifikationsdauer im Vergleich zur Dauer einer Simulation auf. Das liegt im grundsätzlichen Unterschied zwischen Verifikation und Simulation und der Tatsache, dass bei der Verifikation der ganze Zustandsraum untersucht wird. Als Grundlage für die Spezifikation in dieser Arbeit wurden Testfälle aus der Praxis genommen. Das war der erste Schritt bei der Anwendung des Verfahrens, mit dem man gezeigt hat, dass alle nötigen Transformationen für die Anwendung formaler Verifikation in der Praxis automatisierbar sind. Eine andere, im Bezug auf den ganzen Zustandsraum eines Systems formulierte, Spezifikation wird allerdings für die formale Verifikation von Vorteil. Im Moment bezieht sich Systemspezifikation auf konkrete Belegungen aller Variablen. Andererseits kann man bei der Verifikation auf Basis einer Spezifikation, bei der nicht alle Parameter vordefiniert

sind, ein richtiges Systemverhalten verlangen. Solche Spezifikation muss von Domainexperten erstellt werden.

Neben den vielen positiven Aspekten hat das vorgestellte Verifikationsverfahren einige Einschränkungen, die bei der Weiterentwicklung des Ansatzes berücksichtigt werden sollen:

- Die Werkzeugunterstützung für das Verfahren ist noch nicht ausgereift. Bevor das Verfahren in der Praxis eingesetzt wird, müssen die Werkzeuge ausgerüstet werden. Dazu gehört sowohl die Gewährleistung der Stabilität aller Transformationen als auch Entwicklung einer geeigneten graphischen Benutzeroberfläche.

- Das Verfahren wurde für die vorgestellte Softwarekomponente entwickelt. Damit unterstützt es eine Teilmenge aller FUP-Operatoren. Der Ansatz sollte vervollständigt werden, indem eine Unterstützung für alle FUP-Operatoren angeboten wird.

- In der vorliegenden Arbeit wurde ein Verfahren für die FUP-Verifikation vorgeschlagen. Mit der vorgestellten Fallstudie wurde die Vorgehensweise der Verifikation veranschaulicht. Dabei wurde die Spezifikation anhand der existierenden Testfälle konstruiert. Aus dem oben genannten Grund, dass eine neue, für die Verifikation geeignete, Spezifikation erstellt werden sollte, wurden nicht alle Testfälle (die es mehr als 100 gibt) in die Logik umgesetzt.

- Die vorgestellte Softwarekomponente Weiche ist nur eine Komponente der Stellwerksoftware. Mit deren Verifikation ist man immer noch weit von der Verifikation der ganzen Stellwerksoftware entfernt. Dazu kann das Konzept der kompositionalen Verifikation beitragen, indem das Konzept an das vorgestellte Verfahren angepasst wird. Mehr darüber findet man im Ausblick im folgenden Kapitel.

7 Abschließende Betrachtungen

In diesem Kapitel werden die wesentlichen Punkte und die Ergebnisse dieser Arbeit zusammengefasst. Das Kapitel schließt mit einem Ausblick auf zukünftige Arbeiten auf diesem Gebiet.

7.1 Zusammenfassung

In der vorliegenden Arbeit wurde ein Verfahren für die Verifikation von SPS-Software präsentiert. Das Verfahren wurde an der Steuerung SIMATIC S7 des Herstellers Siemens betrachtet. Das vorgestellte Verifikationsverfahren lässt sich auf die grafische Sprache FBS anwenden. Da die Implementierung dieser Sprache bei SIMATIC S7 FUP genannt wird, wurde zuvor von FUP-Verifikation gesprochen. Die wichtigsten Punkte der Arbeit werden im Folgenden zusammengefasst.

7.1.1 Formalisierung von FUP

Als Verifikationsmethode wurde in der Arbeit Model Checking verwendet. Um damit die Korrektheit eines Systems zu überprüfen, muss das System eine geeignete Struktur haben. Beispielsweise ist ein Transitionssystem geeignet für die Anwendung formaler Verifikation.

Aus dem Grund wurde der Schwerpunkt in der ersten Phase des Projekts auf Formalisierung der FUP-Sprache gesetzt. Dabei wurde sowohl die Syntax als auch die Semantik der Sprache formal beschrieben. Als Ergebnis konnte jedem FUP-Programm ein Transitionssystems zugewiesen werden. Das Ergebnis war von großer Bedeutung für die weiteren Transformationsschritte auf dem Weg von einem FUP-Programm zu einem für das Model Checking geeigneten Modell. Aufgrund dessen war man in der Lage die Äquivalenz zwischen Modellen in der durchgeführten Transformationen formal zu zeigen.

Die erzielte FUP-Formalisierung bietet eine Grundlage für die Anwendung weiterer formalen Methoden. In dieser Arbeit wird das Model Checking durch einen konkreten Model Checker durchgeführt. Die geschaffte Formalisierung der Sprache kann allerdings ein guter Startpunkt für viele weitere Arbeiten sein.

7.1.2 Textuelle Darstellung von FUP

Wie vorher erwähnt, ist zur Zeit die formale Verifikation in der Industrie noch nicht weit verbreitet. Es ist nur eine Frage der Zeit, bis dies zumindest bei sicherheitsrelevanten Anwendungen ein Standard wird. Immerhin die textuelle Darstellung eines FUP-Programms ist ein Zwischenschritt in dem vorgestellten Verifikationsverfahren mit einem Potential, in der Praxis umgehend angewendet zu sein.

Ein wichtiger Schritt zwischen Formalisierung der FUP-Sprache und der Verifikation von FUP-Programmen ist die Transformation der grafischen FUP-Programmen in einer textuellen Form. Diese Darstellung, die tFUP genannt wird, bietet einige Analysemöglichkeiten für FUP-Programme.

Um korrekte Funktionalität eines FUP-Programms nachzuweisen, kann in der Praxis ein von den folgenden zwei Vorgehensweisen verfolgt werden:

- Man kann das Programm auf einer echten oder einer simulierten SPS laufen lassen (testen); oder
- Man kann versuchen, den in FUP abgelegten Algorithmus auf eine höhere Abstraktionsebene darzustellen, die dann leichter analysierbar ist.

Die zweite Vorgehensweise wird durch den Programmierstil von FUP erschwert. Ein FUP-Programm hat nämlich keine strukturierte Form. Beispielsweise werden Programmläufe durch bedingte Sprünge entschieden (der berühmte *goto*-Befehl). Ein *if*- oder ein *while*-Befehl steht nicht zur Verfügung. Ein größeres FUP-Programm mit mehreren Dutzend Netzwerken, die womöglich auch noch komplexer Natur sind, entzieht sich also einer strafferen algorithmischen Darstellung. Dieser Umstand macht das Erkennen eines Zustandsautommaten im Programm schwierig, denn der Zustandsautomat verteilt sich inklusive vielen Sprüngen auf viele Netzwerke. Viele Anwendungen sind im Kern aber nichts anderes als Zustandsautomaten.

Mit der eingeführten textuellen Darstellung eines FUP-Programms kann eine Analyse des verborgenen Automaten betrieben werden. Damit kann tFUP eine

Antwort auf viele Validierungsfragen geben, die durch reines Durchlesen eines Programms nicht erkannt werden und aufgrund der bei der SPS fehlenden Testüberdeckungs-Möglichkeiten nicht auffallen. Zusätzlich bietet tFUP eine Möglichkeit diese Syntax in eine weitere Syntax zu transformieren, für die bereits ausreichende Analysewerkzeuge existieren.

7.1.3 Verifikationsverfahren

In den letzten Jahrzehnten wurden erhebliche Fortschritte in der formalen Verifikation von SPS-Software erreicht. Jedoch gibt es immer noch kein standardisiertes Vorgehen zur SPS-Verifikation. Wenn man die veröffentlichten Studien betrachtet, lässt sich feststellen, dass man sich bisher deutlich mehr mit der Verifikation von Textsprachen als mit der Verifikation von grafischen Sprachen beschäftigt hat. Die meist untersuchte Programmiersprache dabei ist AWL. Demgegenüber wurde FUP noch nicht so viel erforscht. Um diese Lücke zu schließen, wird in der vorliegenden Arbeit ein Verifikationsverfahren für die Verifikation von FUP-Programmen vorgestellt. Die Verifikation wird mit Hilfe des Model Checker NuSMV durchgeführt. Mit dem Verfahren wird für ein FUP-Programm das entsprechende NuSMV-Modell in drei Schritten erstellt:

1. Zuerst wird das Programm im textuellen Format textFUP dargestellt, in dem jeder graphische FUP-Operator durch eine textuelle Anweisung repräsentiert ist.

2. Danach wird eine Vereinfachung des textFUP-Formats vorgenommen. Dies wird im tFUP-Format gespeichert.

3. Anschließend wird aus der tFUP-Darstellung das entsprechende NuSMV--Modell erstellt.

Da sich damit auch F-FUP-Programme verifizieren lassen, kann mit dem Verfahren auch die Korrektheit einer Anwendungsapplikation für sicherheitsrelevante Systeme auf SPS-Basis bewiesen werden.

7.1.4 Einsatz in der Industrie

Die Anwendung des vorgestellten Verifikationsverfahrens wurde in der Arbeit mit einer Fallstudie veranschaulicht, die aus dem Bereich der Eisenbahnautomatisierung

kommt. Konkret wurde die FUP-Software betrachtet, die bei einer Stellwerkfamilie benutzt wird. Die betrachtete Stellwerksoftware ist in mehrere Komponenten aufgeteilt, wobei jede Komponente für die Bearbeitung einzelner Stellwerkfunktionen zuständig ist. In der Fallstudie wurde ein Modul der Komponente Weiche betrachtet, das für die Weichensteuerung zuständig ist.

In der vorliegenden Arbeit wird gezeigt, dass die formale Verifikation in der Lage ist, im Gebiet der Eisenbahntechnik zum Einsatz zu kommen. Darüber hinaus wird ein Tool gebaut, mit dem man FUP-Programme im Allgemeinen, also über die konkrete Fallstudie verifizieren kann. Zusätzlich zur automatischen Überführung des FUP-Programms in das für Model Checking geeignete Modell wird in der vorgestellten Fallstudie auch die Spezifikation automatisch in eine temporale Logik überführt. Damit wird der ganze Prozess der formalen Verifikation vollautomatisiert.

7.1.5 Kritikpunkte

Wie am Ende vorheriges Kapitels bereit gesagt wurde, hat das vorgestellte Verifikationsverfahren neben den vielen positiven Aspekten auch einige Einschränkungen, die bei der Weiterentwicklung des Ansatzes berücksichtigt werden sollten:

- Die Werkzeugunterstützung für das Verfahren ist noch nicht ausgereift. Bevor das Verfahren in der Praxis eingesetzt wird, müssen die Werkzeuge sowohl die Gewährleistung der Stabilität aller Transformationen als auch Entwicklung einer geeigneten grafischen Benutzeroberfläche bieten.

- Das Verfahren wurde für die vorgestellte Softwarekomponente entwickelt. Damit unterstützt es eine Teilmenge aller FUP-Operatoren. Der Ansatz sollte vervollständigt werden, indem eine Unterstützung für alle FUP-Operatoren angeboten wird.

- In der vorgestellten Fallstudie wurde die Spezifikation anhand der existierenden Testfällen konstruiert. Aus dem Grund, dass eine neue für die Verifikation geeignete Spezifikation erstellt werden sollte, wurden nicht alle Testfälle, von denen es mehr als 100 gibt, in die Logik umgesetzt.

- Die vorgestellte Softwarekomponente Weiche ist nur eine Komponente der Stellwerksoftware. Mit deren Verifikation ist man immer noch weit von der Verifikation der ganzen Stellwerksoftware entfernt. Eine Lösung dafür liegt in

der kompositionalen Verifikation, die im folgenden Abschnitt näher erläutert wird.

7.2 Ausblick

Das vorgestellte Verifikationsverfahren kann in vielerlei Hinsicht weiterentwickelt werden, was im Folgenden kurz vorgestellt wird. Abhängig davon, ob die Aspekte allgemeingültig sind oder einen Bezug zum hier vorgestellten Projekt aus der Eisenbahnautomatisierung haben, werden die Vorschläge für die Weiterentwicklung des Verfahrens im Folgenden in zwei Abschnitte präsentiert.

7.2.1 Projektbezogene Ansätze

Wenn man das Verifikationsverfahren im Zusammenhang mit der in der Fallstudie vorgestellten Softwarekomponente und der übergreifenden Stellwerksoftware betrachtet, können folgende Punkte in Zukunft erarbeitet werden.

Bounded Model Checking

Neben dem herkömmlichen BDD-basierten Model Checking Prinzip kann das Model Checking Verfahren auch auf die SAT (satisfiability) Techniken basieren. Hinter dem SAT-Problem verbirgt sich die Frage, ob eine aussagenlogische Formel erfüllbar ist. Die Grundidee des SAT-basierten Model Checking ist, Gegenbeispiele einer bestimmten, vorher festgelegte Länge zu betrachten. Beim Bounded Model Checking (BMC) wird eine Grenze bestimmt, die die maximale Länge des Gegenbeispiels aufweist.

BMC hat folgende Vorteile ([BCCZ99]):

- Gegenbeispiele werden schnell gefunden;
- Die Gegenbeispiele der minimalen Länge werden gefunden; und
- BMC braucht deutlich weniger Speicher als BDD-basierte Verfahren.

Mit dem in dieser Arbeit vorgestellten Verifikationsverfahren werden *smv*-Dateien generiert. Einer von den weiteren Schritten in diesem Projekt könnte sein, die Verifikationsmöglichkeiten von den *smv*-Dateien durch BMC zu untersuchen. Dies kann beispielsweise mit Hilfe der folgenden Werkzeugen gemacht werden:

- NuSMV - Die aktuelle Version von NuSMV (NuSMV 2.4.3) unterstützt das SAT-basierte BMC von LTL-Spezifikation. Die Unterstützung für CTL sollte in folgenden Versionen zur Verfügung gestellt werden.

- EBMC (the Enhanced Bounded Model Checker) - Für das *smv*-Format wird BMC auch vom Model Checker EBMC angeboten. Im Gegensatz zu NuSMV, der für BMC erweitert wird, wurde EBMC vollkommen für BMC konstruiert (siehe [KP]).

In SPS-Domänen wurden SAT-Solvers bereit in [SD08] benutzt. In der Arbeit wurde das Verhalten eines Anwenderprogramms in Bezug auf verschiedene SPS in mehrere SystemC-Modelle erfasst und anschließend der Äquivalenzvergleich von Modellen durchgeführt.

Neuformulierung der Spezifikation

Die in der Arbeit vorgestellten Verifikationsergebnisse weisen deutlich höhere Werte bezüglich der Verifikationsdauer im Vergleich zur beispielsweise Dauer der Testausführung, die zur Zeit in der Praxis angewendet wird. Das liegt an der Tatsache, dass bei der Verifikation jeder einzelne Zustand im ganzen Zustandsraum auf die Gültigkeit der Spezifikation überprüft wird.

Als Grundlage für die Spezifikation in dieser Arbeit wurden Testfälle aus der Praxis genommen. Damit wurde gezeigt, dass alle nötige Transformationen für die Anwendung formaler Verifikation in der Praxis automatisierbar sind. Andererseits bezieht sich im Moment die Systemspezifikation auf konkrete Variablenbelegungen. Hingegen kann man bei der Verifikation auf Basis einer Spezifikation, bei der nicht alle Parameter vordefiniert sind, ein richtiges Systemverhalten verlangen. Eine Spezifikation dieser Art, die im Bezug zum ganzen Zustandsraum eines Systems formuliert wird, wird für die formale Verifikation von Vorteil. Solche Spezifikationen müssen von Domainexperten erstellt werden.

Kompositionale Verifikation

Das vorgestellte Verifikationsverfahren kann weiterentwickelt werden, indem die Aspekte der kompositionalen Verifikation eingebaut werden. Mehr Informationen über kompositionale Verifikation findet man in [Pin02]. Ein Beispiel ihrer Einsatz im SPS-Domain wurde in [Luk05] repräsentiert.

In der vorliegenden Arbeit wurde die Komponente Weiche der Stellwerksoftware betrachtet. Analog kann auch beispielsweise die Komponente Signal betrachtet werden. Verifikation einzelner Komponente ist ein wichtiger Schritt zur Verifikation der ganzen Software. Auf den ersten Blick scheint Stellwerksoftware geeignet für die Anwendung der kompositionalen Verifikation zu sein. In [Ost09]) wurden einige Komponenten dieser Software und deren Kommunikation abstrahiert und manche allgemeine Aspekte verifiziert. Damit wurde ein Einblick in die Problematik erreicht. Ein genaueres Konzept muss aber noch erarbeitet werden.

7.2.2 Allgemeingültige Ansätze

Allgemein betrachtet, können auch folgende Aspekte beim Verifikationsverfahren berücksichtigt werden.

Theorem Proving

Die Größe der vorgestellten Fallstudie entspricht genau der Leistungsstärke des Model Checking. Andere Komponente der betrachteten Software können durchaus umfangreicher sein. Von daher muss man sich bei kompositionaler Verifikation bewusst sein, dass bevor mehrere Komponente verifiziert werden, das ganze System abstrahiert werden muss.

Eine andere Möglichkeit ist, die Verifikationstechniken Model Checking und Theorem Proving zu kombinieren (siehe [MN95]). Damit können die Vorteile beider Techniken ausgenutzt werden:

- Der Vorteil von Model Checking ist die hohe Automatisierungsgrad. Allerdings lässt sich die Methode nur auf endliche Systeme anwenden.

- Andererseits wird beim Theorem Proving die Interaktion mit dem Benutzer benötigt. Der Vorteil des Theorem Proving ist, dass damit sowohl endliche als auch unendliche Systeme behandelt werden können.

Die Hauptidee bei der Kombination der beiden Techniken ist, Theorem Proving auf das ganze System anzuwenden und daraus kleinere Systeme zu gewinnen, die danach für Model Checking geeignet sind.

Verifikation von KOP-Programmen

Der Hauptpunkt in dem Verifikationsverfahren ist die Generierung der textuellen Repräsentation des grafischen FUP-Programms. Dies wird anhand von der AWL-Repräsentation des FUP-Programms und der vorgestellten kontextfreien Grammatik geleistet. Es ist vorzustellen, dass ein ähnliches Prinzip sich auch auf KOP-Programme anwenden lässt. Damit könnte ein Verfahren für die KOP-Verifikation zur Verfügung gestellt werden.

Soft-SPS

Zum Abschluss können auch Soft-SPS erwähnt werden. Deren Konzept wurde in Kapitel 2 eingeführt. Mit dem Einsatz von Soft-SPSen ist eine erhebliche Kostensenkung in der Infrastruktur verbunden. Obwohl sie nicht weitverbreitet sind, gewinnen diese Steuerungen langsam an Bedeutung. Obwohl es im Moment noch schwer vorstellbar ist, werden diese Steuerungen vielleicht eines Tages ein Standard in der Industrie sein. Da diese Steuerungen auch mit der Standard SPS-Software programmierbar sind, lässt sich das vorgestellte Verifikationsverfahren dann auch auf die Soft-SPS anwenden.

A Grammatik

Mithilfe der folgenden kontextfreien Grammatik wird das AWL-Format eines FUP-Programms in textFUP umgewandelt. Hier werden alle FUP-Anweisungen berücksichtigt, die für die Implementierung der Softwarekomponente Weiche benötigt sind. Ein kleiner Auschnitt aus der Grammatik wurde in Kapitel 6 und Tabelle 6.4 präsentiert. In dem entsprechenden Abschnitt „Erstellen des textFUP-Formats" wurde die ganze Vorgehensweise genauer beschrieben.

<Programm> ::= FUNCTION_BLOCK TITLE = <ID> <Deklaration>
<Struktur> END_FUNCTION_BLOCK

<Deklaration> ::= <Kommentare> AUTHOR : <ID> FAMILY : <ID>
NAME : <ID> VERSION : <Nummer> . <Version>
CODE_VERSION1 <VarAbschnitt>

<ID> ::= <Buchstabe> <BuchstabeZifferN>

<Buchstabe> ::= A | B | C | D | ... | Z | a | b | ... | z

<BuchstabeZifferN> ::= | <Buchstabe> <BuchstabeZifferN>
| <Ziffer0> <BuchstabeZifferN>

<Ziffer0> ::= 0 | <Ziffer>

<Ziffer> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ZifferN> ::= <Ziffer> | <Ziffer> <ZifferN>

<Nummer> ::= <Ziffer> | <Ziffer-0> <ZifferN>

<Operand> ::= <Var> | <Nummer>

<Var> ::= # <ID> | «ID> ". <ID>

<Ende> ::= ; <KommentarN>

<KommentarN> ::= | <KommentarN> <Kommentar>

<Kommentar> ::= \\ <BuchstabeZifferN>

```
<VarAbschnitt>      ::= <VarInput> <VarOutput> <VarInOut> <VarTemp>
                        <VarStat>
<VarInput>          ::= VAR_INPUT <VarDeklarationN> END_VAR
<VarOutput>         ::= VAR_OUTPUT <VarDeklarationN> END_VAR
<VarInOut>          ::= VAR_IN_OUT <VarDeklarationN> END_VAR
<VarTemp>           ::= VAR_TEMP <VarDeklarationN> END_VAR
<VarStat>           ::= VAR <VarDeklarationN> END_VAR
<VarDeklarationN>   ::=   | <VarDeklarationN> <VarDeklaration>
<VarDeklaration>    ::= <ID> : BOOL <BoolZuweisung> <Ende>
                      | <ID> : INT <IntZuweisung> <Ende>
                      | <ID> : WORD <IntZuweisung> <Ende>
                      | <Kommentar>
<BoolZuweisung>     ::=   | := TRUE | := FALSE
<IntZuweisung>      ::=   | := <Nummer>
<Struktur>          ::= BEGIN <NetzwerkN>
<NetzwerkN>         ::=   | <NetzwerkN> <Netzwerk>
<Netzwerk>          ::= NETWORK <Titel> <Kommentare> <Anweisung>
                      | NETWORK <Titel> <Kommentare> <ID> : <Anweisung>
<Anweisung>         ::=   | <Bitlogik> <BitlogikEnde>
                      | <Vergleich> = <Var> <Ende> <ZuweisungEnde>
                      | <Vergleich> <Sprung> <ID> <Ende>
                      | <Bitlogik> S <Var> <Ende> <Bitlogik> R <Var>
                      <Ende>
                      | L <Operand> <Ende> T <Var> <Ende>
                      | <IntMath>
                      | <Sprung> <ID> <Ende>
                      | <Call>
                      | <MemZuweisung> <Call>
                      | <Bitlogik> = <MemBit> <Ende> <BitlogikMemEnde>
<Bitlogik>          ::= U <Operand> <Ende>
                      | UN <Operand> <Ende>
                      | O <Operand> <Ende>
                      | ON <Operand> <Ende>
                      | X <Var> <Ende>
```

| U(<Ende> <Vergleich>) <Ende>
| O(<Ende> <Vergleich>) <Ende>
| U(<Ende> <Bitlogik>) <Ende>
| O(<Ende> <Bitlogik>) <Ende>
| X(<Ende> <Bitlogik>) <Ende>
| <Bitlogik> U <Operand> <Ende>
| <Bitlogik> UN <Operand> <Ende>
| <Bitlogik> O <Operand> <Ende>
| <Bitlogik> ON <Operand> <Ende>
| <Bitlogik> X <Operand> <Ende>
| <Bitlogik> NOT <Ende>
| <Bitlogik> O <Ende> <Bitlogik>
| <Bitlogik> U(<Ende> <Vergleich>) <Ende>
| <Bitlogik> O(<Ende> <Vergleich>) <Ende>
| <Bitlogik> = <Var> <Ende> U <Var> <Ende>
| <Bitlogik> FP <Var> <Ende>
| <Bitlogik> FN <Var> <Ende>
| <Bitlogik> U(<Ende> <Bitlogik>) <Ende>
| <Bitlogik> O(<Ende> <Bitlogik>) <Ende>

<Vergleich> ::= L <Operand> <Ende> L <Operand> <Ende>
COMPARETOK <Ende>

<BitlogikEnde> ::= <SetResetEnde> | <ZuweisungEndeN>
| <SprungEnde> | <SaveEnde>

<SetResetEnde> ::= S <Var> <Ende> <SetResetEnde0>
| R <Var> <Ende> <SetResetEnde0>

<SetResetEnde0> ::= | <SetResetEnde0> S <Var> <Ende>
| <SetResetEnde0> R <Var> <Ende>

<ZuweisungEndeN> ::= = <Var> <Ende> <ZuweisungEnde>

<ZuweisungEnde> ::= | <ZuweisungEnde> = <Var> <Ende>

<SprungEnde> ::= <Sprung> <ID> <Ende>

<Sprung> ::= SPA | SPB | SPBN

<SaveEnde> ::= SAVE <Ende> BEB <Ende>

<Call> ::= CALL «ID> "(<ArgumentN>) <Ende>

<ArgumentN> ::= <ID> := <Operand>

| <ID> := <MemBit>
| <ArgumentN> , <ID> := <Operand>
| <ArgumentN> , <ID> := <MemBit>

<MemBit> ::= L <Ziffer> <Ziffer> . <Ziffer>

<MemZuweisung> ::= <Bitlogik> = <MemBit> <Ende> BLD 103 <Ende>
| <MemZuweisung> <Bitlogik> = <MemBit> <Ende> BLD 103 <Ende>

<BitlogikMemEnde> ::= U <MemBit> <Ende> <Bitlogik> <bld> <SetResetEnde>
| <BitlogikMemEnde> U <MemBit> <Ende> <bld> R <Var> <Ende>

<bld> ::= | BLD 102 <Ende>

<IntMath> ::= L <Operand> <Ende> L <Operand> <Ende> -I <Ende> T <Var> <Ende>
| L <Operand> <Ende> L <Operand> <Ende> +I <Ende> T <Var> <Ende>
| L <Operand> <Ende> L <Operand> <Ende> *I <Ende> T <Var> <Ende>

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Asp05] Aspern von, Jens: *SPS Grundlagen: Aufbau, Programmierung (IEC 61131, S7), Simulation, Internet, Sicherheit.* Hüthig GmbH & Co. KG Heidelberg, 2005. – ISBN 978–3778540602

[BB89] Bläsius, Karl Hans ; Bürckert, Hans-Jürgen: *Deduction Systems in Artificial Intelligence.* Halsted Press: a division of John Wiley and Sons, 1989 (Ellis Horwood Series in Artificial Intelligence)

[BCCZ99] Biere, Armin ; Cimatti, Alessandro ; Clarke, Edmund ; Zhu, Yunshan: *Symbolic Model Checking without BDDs.* 1999

[BK08] Baier, Christel ; Katoen, Joost-Pieter: *Principles of Model Checking.* MIT Press, 2008

[BMF02] Brinksma, Ed ; Mader, Angelika ; Fehnker, Ansgar: Verification and optimization of a PLC control schedule. In: *STTT* 4 (2002), Nr. 1, S. 21–33

[Bra05] Braband, Jens: *Risikoanalysen in der Eisenbahn-Automatisierung.* Eurailpress, 2005. – ISBN 978–3777103358

[CC79] Cousot, P. ; Cousot, R.: Systematic design of program analysis frameworks. In: *In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1979, S. 269–282

[CCGR00] Cimatti, A. ; Clarke, E. ; Giunchiglia, F. ; Roveri, M.: NUSMV: a new symbolic model checker. In: *International Journal on Software Tools for Technology Transfer* 2 (2000)

[CCJ+] Cavada, R. ; Cimatti, A. ; Jochim, Ch. A. ; Keighren, G. ; Olivetti, E. ; Pistore, M. ; Roveri, M. ; Tchaltsev, A. ; CMU and ITC-irst (Hrsg.): *NuSMV 2.4 User Manual*. http://www.nusmv.irst.itc.it: CMU and ITC-irst

[CCK+] Cavada, R. ; Cimatti, A. ; Keighren, G. ; Olivetti, E. ; Pistore, M. ; Roveri, M. ; CMU and ITC-irst (Hrsg.): *NuSMV 2.2 Tutorial*. http://www.nusmv.irst.itc.it: CMU and ITC-irst

[CCLP00] Canet, G. ; Couffin, S. ; Lesage, J. j. ; Petit, A.: Towards the automatic verification of PLC programs written in Instruction List. In: *IEEE International Conference on Systems, Man and Cybernetics*, 2000, S. 2449–2454

[CE81] Clarke, E.M. ; Emerson, E.A.: Characterizing Properties of Parallel Programs as Fixpoints. In: *7th International Colloquium on Automata, Languages and Programming* Bd. 85 of Lecture Notes in Computer Science, Springer-Verlag, 1981

[CES86] Clarke, E.M. ; Emerson, E.A. ; Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. In: *ACM Transactions on Programming Languages and Systems*, 1986, S. 244–263

[CGL93] Clarke, E.M. ; Grumberg, O. ; Long, D.: Verification tools for finite-state concurrent systems. In: *A Decade of Concurrency* Bd. 803 in LNCS, Springer Verlag, 1993, S. 124–175

[CGP99] Clarke, Edmund M., Jr ; Grumberg, Orna ; Peled, Doron A.: *Model Checking*. MIT Press, 1999

[Chr95] Christgau, Martin: *Bisimulationen und Äquivalenzbegriffe für Transitionssysteme und Ereignisstrukturen*, Universität Mannheim, Diplomarbeit, 1995

[Cou78] Cousot, Patrick: *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*, Université scientifique et médicale de Grenoble, Diss., 1978

[Die97] Dierks, Henning: PLC-automata: A new class of implementable real-time automata. In: *ARTS'97, volume 1231 of LNCS*, Springer-Verlag, 1997, S. 111–125

[Die98] Dierks, Henning: Comparing model-checking and logical reasoning for real-time systems. In: *Formal Aspects of Computing* (1998)

[Feh89] Fehr, Elfriede: *Semantik von Programmiersprachen.* Springer-Verlag, 1989. – ISBN 978–3540151630

[Fig06] Figura, Christian: *Überdeckungstests für fehlersichere Funktionspläne auf Basis einer geeigneten Überführung*, Martin-Luther-Universität Halle-Wittenberg, Diplomarbeit, 2006

[FL98] Frey, Georg ; Litz, Lothar: Verification and Validation of Control Algorithms by Coupling of Interpreted Petri Nets. In: *IEEE SMC*, 1998, S. 7–12

[FL99] Frey, Georg ; Litz, Lothar: Methoden und Werkzeuge zum industriellen Steuerungsentwurf - Historie, Stand, Ausblick. In: *at 4/99*, 1999, S. 145–156

[Gie05] Giessler, Walter: *SIMATIC S7 SPS-Einsatzprojektierung und -programmierung.* VDE Verlag GMBH, 2005. – ISBN 978–3800728893

[GSF06] Gourcuff, Vincent ; Smet, Olivier D. ; Faure, Jean-Marc: Efficient representation for formal verification of PLC programs. In: *Proc. of the 8th International Workshop on Discrete Event Systems*, 2006, S. 182–187

[GSF08] Gourcuff, Vincent ; Smet, Olivier D. ; Faure, Jean-Marc: Improving large-sized PLC programs verification using abstractions. In: *17th IFAC World Congress*, 2008

[Hec77] Hecht, M. S.: *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977

[HIZ98] Hassapis, G. ; I.Kotini ; Z.Doulgeri: Validation of a SFC software specification by using hybrid automata. In: *IN-COM*, 1998, S. 65–70

[HLB03] Huuck, Ralf ; Lukoschus, Ben ; Bauer, Nanette: A Model-Checking Approach to Safe SFCs. In: *IMACS Multiconference on Computational Engineering in Systems Applications (CESA 2003)*, 2003

[HM98] Heiner, Monika ; Menzel, Thomas: A Petri net semantics for the PLC language Instruction List. In: *Proc. of the International Workshop on Discrete Event Systems (WoDES)*, 1998, S. 161–166

[Hoc06] Hochschule Bremerhaven (Hrsg.): *Steuerungs- und Feldbustechnik, Unterlagen zur Lehrveranstaltung.* Hochschule Bremerhaven, 2006

[HPP+99] Hanisch, Hans-Michael ; Pannier, Thorsten ; Peter, Dirk ; Roch, Stephan ; Starke, Peter: Modeling and Formal Verification of a Modular Level-Crossing Controller Design. In: *Automatisierungstechnik* Bd. 47, Oldenbourg Verlag, 1999, S. 366–373

[HR00] Huth, Michael ; Ryan, Mark: *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2000

[Huu03] Huuck, Ralf: *Software Verification for Programmable Logic Controllers*, Christian-Albrechts-University of Kiel, Diss., 2003

[Huu04] Huuck, Ralf: Semantics and Analysis of Instruction List Programs. In: *Proc. of the Second Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004)*, Electronic Notes in Theoretical Computer Science 115, Elsevier, 2004, S. 3–18

[INR09] INRIA (Hrsg.): *The Coq Proof Assistant - Reference Manual.* : INRIA, 2009

[Int00] International Electrotechnical Commission: *International Standard 61508, Functional safety of electrical/elctronic/programmable electronic safety-related systems.* 2000

[Int03] International Electrotechnical Commission: *International Standard 61131-3, Programmable controllers - Part 3: Programming languages.* 2003. – ISBN 2–8318–6653–7

[Jeo06] Jeon, Seungjae: *Verification of Function Block Diagram through Verilog Translation*, Korea Advanced Institute of Science and Technology, Diplomarbeit, 2006

[Kat99] Katoen, Joost-Pieter: *Concepts, Algorithms, and Tools for Model Checking.* Universität Erlangen-Nürnberg, 1999

[Kin05] Kindler, Ekkart: *Semantik - Skript zur Vorlesung Semantik von Programmiersprachen an der Universität Paderborn.* 2005

[Kin08] Kinder, Sebastian: *Automated Validation and Verification of Railway Specific Components and Systems*, Universität Bremen, Diss., 2008

[KJJ+08] Koh, Kwang Y. ; Jee, Eun K. ; Jeon, Seung J. ; Seong, Poong H. ; Cha, Sung D.: A Formal Verification Method of Function Block Diagrams with Tool Supporting: Practical Experiences. In: *Annals of DAAAM for 2008 & Proceedings of the 19th International DAAAM Symposium*, 2008

[KP] Kroening, Daniel ; Purandare, Mitra: *EBMC.* `http://www.cprover.org/ebmc/`. – Last visited 27.10.2009

[KP96] Kowalewski, Stefan ; Preussig, Jörg: Verification of sequential controllers with timing functions for chemical processes. In: *13th IFAC World Congress*, 1996

[Kri63a] Kripke, S.A.: Semantical Analysis of Modal Logic I – Normal Modal Propositional Calculi. In: *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* (1963), S. 67–96

[Kri63b] Kripke, S.A.: Semantical Considerations on Modal Logic. In: *Acta Philosophica Fennica – Modal and Many-values Logics* (1963), S. 83–94

[KSJ+07] Koh, Kwang Y. ; Seong, Poong H. ; Jee, Eun K. ; Jeon, Seung J. ; Park, Gee Y. ; Kwon, Kee-Choon: A Formal Verification Method of Function Block Diagram. In: *Transactions of the Korean Nuclear Society Spring Meeting*, 2007

[Lit05] Litz, Lothar: *Grundlagen der Automatisierungstechnik: Regelungssysteme- Srezerungssysteme- Hybride Systeme.* Oldenbourg Wissenschaftsverlag, 2005. – ISBN 978–3486273830

[Luk05] Lukoschus, Ben: *Compositional Verification of Industrial Control Systems*, Christian-Albrechts-University of Kiel, Diss., 2005

[LYF05] Loeis, Kingliana ; Younis, Mohammed B. ; Frey, Georg: Application of Symbolic and Bounded Model Checking to the Verification of Logic Control Systems. In: *IEEE International Conference on Emerging Technologies and Factory Automation* Bd. 1, 2005, S. 247–250

[Mad00] Mader, Angelika: A classification of PLC models and applications. In: *In WODES 2000: 5th Workshop on Discrete Event Systems*, 2000, S. 21–23

[MBWB01] Mader, Angelika ; Brinksma, Ed ; Wupper, Hanno ; Bauer, Nanette: *Design of a PLC Control Program for a Batch Plant VHS Case Study 1.* 2001

[MF01] Mertke, Thomas ; Frey, Georg: Formal verification of PLC-programs generated from signal interpreted petri nets. In: *IEEE SMC*, 2001, S. 2700–2705

[MM00] Mertke, Thomas ; Menzel, Thomas: Methods and Tools to the verification of safety-related control software. In: *IEEE SMC*, 2000, S. 2455–2457

[MN95] Müller, Olaf ; Nipkow, Tobias: Combining Model Checking and Deduction for I/O-Automata. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer-Verlag, 1995, S. 1–16

[MW99] Mader, Angelika ; Wupper, Hanno: Timed automaton models for simple programmable logic controllers. In: *Proc. of the Euromicro Conference on Real-Time Systems*, IEEE Computer Society, 1999, S. 114–122

[NAS97] NASA: *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems. Vol. 2: A Practitioner's*

Companion. NASA Office of Safety and Mission Assurance, Washington D.C., Report NASA-GB-001-97, 1997

[NPW09] Nipkow, Tobias ; Paulson, Lawrence C. ; Wenzel, Markus: *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2009

[Ost09] Ostwald, Maico: *Analyse der Anwendung kompositionaler Verifikation von Stellwerkssoftware*, Technische Universität Braunschweig, Diplomarbeit, 2009

[Pac08] Pachl, Jörn: *Systemtechnik des Schienenverkehrs. Bahnbetrieb planen, steuern und sichern*. Vieweg+Teubner, 2008. – ISBN 978–3835101913

[Pin02] Pinger, Ralf: *Kompositionale Verifikation nebenläufiger Softwaremodelle Durch Model Checking*, Technische Universität Braunschweig, Diss., 2002

[Pnu77] Pnueli, Amir: The temporal logic of programs. In: *Proc. 18th Symposium on Foundations of Computer Science*, 1977, S. 46–57

[PPK07] Pavlovic, Olivera ; Pinger, Ralf ; Kollmann, Maik: Automation of Formal Verification of PLC Programs Written in IL. In: Beckert, Bernhard (Hrsg.): *Proc. of 4th International Verification Workshop in connection with CADE-21*, CEUR-WS.org, 2007

[PPKE07] Pavlovic, Olivera ; Pinger, Ralf ; Kollmann, Maik ; Ehrich, Hans Dieter: Principles of Formal Verification of Interlocking Software. In: Schnieder, E. (Hrsg.) ; Tarnai, G. (Hrsg.): *Proc. of the 6th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2007)*, GZVB, 2007

[QS82] Queille, J.P. ; Sifakis, J.: Specification and Verification of Concurrent Systems in Cesar. In: *Proc. 5th International Symposium on Programming* Bd. 137 of LNCS, Springer-Verlag, 1982, S. 337–351

[RD02] Roussel, Jean-Marc ; Denis, Bruno: Safety Properties Verification of Ladder Diagram Programs. In: *Européen des Systèmes Automatisés* 36 (2002), S. 905–917

[RS00] Rossi, O. ; Schnoebelen, Ph.: Formal Modeling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs. In: *ADPM'2000*, 2000

[SD08] Sülflow, Andre ; Drechsler, Rolf: Verification of PLC Programs using Formal Proof Techniques. In: *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008)*, 2008, S. 43–50

[Sie04a] Siemens (Hrsg.): *SIMATIC - Erste Schritte und Übungen mit STEP 7 V5.3.* : Siemens, 2004

[Sie04b] Siemens (Hrsg.): *SIMATIC - Funktionsplan (FUP) für S7-300/400.* : Siemens, 2004

[Sie04c] Siemens (Hrsg.): *SIMATIC - Programmieren mit STEP 7 V5.3.* Siemens, 2004

[Sie09] Siemens AG (Hrsg.): *SIMATIC Controller Die innovative Lösung für alle Automatisierungsaufgaben.* Siemens AG, 2009

[SKS04] Song, Myung J. ; Koo, Seo R. ; Seong, Poong-Hyun: Verification method for the FBD-style design specification using SDT and SMV. In: *IASTED Conf. on Software Engineering*, 2004, S. 206–211

[Son03] Song, Myung J.: *Development of a Verification Method for the FBD-style Design Specification Using ESDT and SMV*, Korea Advanced Institute of Science and Technology, Diplomarbeit, 2003

[SSRSC01] Shankar, N. ; S.Owre ; Rushby, J.M. ; Stringer-Calvert, D.W.J.: *PVS Prover Guide.* SRI International, 2001

[Tau89] Taubner, Dirk: *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets.* Springer-Verlag Berlin Heidelberg, 1989. – ISBN 3–540–51525–9

[VK97] Völker, Norbert ; Krämer, Bernd J.: A highly dependable computer architecture for safety-critical control applications. In: *Real-Time Systems Journal* Bd. 13(3), Springer, 1997, S. 237–251

[VK99] Völker, Norbert ; Krämer, Bernd J.: Modular verification of function block based industrial control systems. In: *Joint 24th IFAC/IFIP Workshop on Real-Time Programming and The Third Workshop on Active and Real-Time Database Systems*, 1999

[Vö98] Völker, Norbert: *Ein Rahmen zur Verifikation von SPS-Funktionsbausteinen in HOL*. Shaker, 1998. – ISBN 382654367X

[Vö00] Völker, Norbert: Towards a HOL Framework for the Deductive Analysis of Hybrid Control Systems. In: *ADPM 2000*, Shaker, 2000

[Wil99] Willems, H.X.: Compact timed automata for PLC programs / University of Nijmegen, Computing Science Institute. 1999. – Forschungsbericht

[WL00] Weng, Xiying ; Litz, Lothar: Verification of logic control design using SIPN and model checking: methods and case study. In: *American Control Conference*, 2000, S. 4072–4076

[YF04] Younis, Mohammed B. ; Frey, Georg: Formalization of PLC Programs to Sustain Reliability. In: *IEEE International Conference on Robotics, Automation and Mechatronics*, 2004, S. 507–511

[ZRK03] Zoubek, Bohumir ; Roussel, Jean-Marc ; Kwiatowska, Marta: Towards Automatic Verification of Ladder Logic Programs. In: *IMACS Multiconference on Computational Engineering in Systems Applications (CESA 2003)*, 2003